UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadores y Automática
(Arquitectura y Tecnología de Computadores e Ingeniería de Sistemas y Automática)

PLANIFICACIÓN DE PROCESOS EN SISTEMAS MULTICORE ASIMÉTRICOS.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Juan Carlos Sáez Alcaide

Bajo la dirección de los doctores

Manuel Prieto
Alexandra Fedorova

Madrid, 2011

ISBN: 978-84-694-2889-4

Planificación de Procesos en Sistemas Multicore Asimétricos

Thread Scheduling on Asymmetric Multicore Systems

Departamento de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

TESIS DOCTORAL

Juan Carlos Sáez Alcaide

Madrid, Diciembre de 2010

Planificación de Procesos en Sistemas Multicore Asimétricos

Memoria presentada por Juan Carlos Sáez Alcaide para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de Manuel Prieto Matías y Alexandra Fedorova.

Thread Scheduling on Asymmetric Multicore Systems

Dissertation submitted by Juan Carlos Sáez Alcaide to the Complutense University of Madrid in partial fulfillment of the requirements for the degree of doctor of philosophy, work supervised by Manuel Prieto Matías and Alexandra Fedorova.

Madrid, 15 de Diciembre de 2010.

A los miembros de mi familia: Regino, Mari Nieves, Javi, Luismi
y, en especial, a Esther.

Agradecimientos

En primer lugar, deseo agracecer de manera sincera a mis directores de tesis Dr. Manuel Prieto y Dra. Alexandra Fedorova. Manuel fue uno de los grandes culpables de que yo comenzara esta tesis doctoral, por lo que le estoy tremendamente agradecido. De él debo destacar su notable inspiración en momentos críticos del desarrollo de este trabajo de investigación, esencial para que esta tesis llegara a un feliz término. Por otra parte, quería también dar las gracias a Sasha por su extraordinaria visión a la hora de centrar el tema de mi tesis, y por su inestimable ayuda prestada durante estos últimos dos años, sobre todo en el complejo proceso de difusión de nuestra investigación. Las ideas propias, siempre enmarcadas en su orientación, experiencia y rigurosidad, han sido la clave del trabajo que mis directores de tesis y yo hemos realizado juntos, el cual no puede concebirse sin su siempre oportuna participación.

Agradezco al Ministerio de Educación y Ciencia (MEC) por la financiación recibida durante estos últimos años a través del programa de Formación de Profesorado Universitario (FPU).

Deseo también dar las gracias a Francisco Tirado, Luis Piñuel y, en general, al grupo de investigación ArTeCS (Group of Architecture and Technology of Computing Systems) de la Universidad Complutense de Madrid por el apoyo prestado durante todo el desarrollo de esta tesis doctoral y por haberme facilitado siempre los medios suficientes para llevar a cabo todas las actividades propuestas durante la misma.

Del mismo modo, quiero agradecer al grupo Synar (Systems, Networking and Architecture) de la Universidad Simon Fraser de Vancouver (Canadá) por brindarme tan excepcional oportunidad de colaboración, así como por haber puesto siempre a mi disposición sus recursos computacionales.

Quiero expresar mi más sincero agradecimiento a Daniel Shelepov por su importante aporte y activa participación durante la primera etapa de mi doctorado. Sus trabajos preliminares me han servido de motivación para construir esta tesis doctoral. Debo destacar, por encima de todo, su rigurosidad y paciencia que hizo que nuestras discusiones, en ocasiones acaloradas, supusieran un notable progreso tanto a nivel científico como personal.

Deseo dar las gracias al Dr. David Koufaty por darme recientemente la oportunidad de acceder a prototipos hardware de Intel y a otros recursos computacionales de acceso restringido. Espero que el futuro nos reserve una fructífera colaboración.

Quería dar las gracias a mis compañeros Edgardo, Rodrigo, David, Rubén, Marco, Fermín, Hugo, Carlos, Darío, Daniel, ... por todos los buenos momentos tanto en el laboratorio como fuera de él. Deseo también agradecer a mis compañeros "canadienses" Sergey B., Sergey Z., Nasser, Ali y Viren por haberme hecho sentir como en casa durante mis estancias en Vancouver.

Deseo agradecer de manera más afectuosa a Mari Nieves, Regino, Javi y Luismi, por estar siempre a mi lado en todo momento y por apoyarme incondicionalmente durante toda mi tesis doctoral. Ellos siempre han creído en mis posibilidades para alcanzar tanto ésta como otras metas que han contribuido a mi enriquecimiento a nivel personal.

Quería dedicar estas últimas líneas para expresar mi más sentido agradecimiento a Esther. Ella es la persona que más directamente ha sufrido las consecuencias del trabajo realizado. A pesar de todo, siempre me ha transmitido paciencia y comprensión, así como el ánimo necesario para encarar las adversidades que encontré en el camino. Nunca le podré estar suficientemente agradecido.

Abstract

Symmetric-ISA (Instruction Set Architecture) asymmetric-performance multicore processors (AMPs) were shown to deliver higher performance per watt and area than its symmetric counterparts [1, 2], and so it is likely that future multicore processors will combine a few *fast* cores characterized by complex pipelines, high clock frequency, high area requirements and power consumption, and many *slow* ones, characterized by simple pipelines, low clock frequency, low area requirements and power consumption.

Recent research has highlighted that efficiency of AMP systems could be improved using two kinds of core specializations [3, 4]. The former ensures that fast cores are used for those applications that efficiently utilize these cores' "expensive" features, while slow cores would be used for applications spending a majority of their execution time stalling the processor, thus utilizing complex cores inefficiently. The latter leverages the effectiveness of these systems by using fast cores to accelerate sequential phases of parallel applications, and devoting slow cores to running parallel phases.

To fully tap into the potential of specialization, the operating system (OS) must be aware of the hardware asymmetry when making scheduling decisions and map applications to cores in consideration of their performance characteristics. While the design and the theoretical benefits of AMPs have been extensively investigated [1, 5], the study of real-world operating system support for these upcoming architectures has not been addressed comprehensively to date. So the questions as to whether this potential can be delivered efficiently by the operating system to unmodified applications, and what the associated overheads are remain open.

In this thesis, we propose a set of OS-level scheduling algorithms aimed to unleash the potential of specialization. These algorithms have been implemented on an actual operating system and extensively evaluated on real multicore hardware made asymmetric via dynamic voltage and frequency scaling (DVFS). Notably, none of these algorithms require changes to applications but only moderate changes to the target operating system, providing proof of concept towards lightweight OS support for asymmetric hardware. Our evaluation also includes an extensive comparison with previously proposed asymmetry-aware schedulers to provide a clearer understanding of the pros and cons behind our proposals.

Contents

Chapter 1

Introduction

In computing systems, the CPU is usually one of the largest consumers of energy. For this reason, reducing CPU power consumption has been a hot topic in the past few years in both the academic community and the industry. To date, one of the most significant challenges in this field was the rise of inherent limitations to the ability to increase performance by driving up clock speed. Faster processor frequencies began to deliver relatively modest performance increases, due to memory access limitations and other issues, while at the same time, power consumption and heat dissipation rose dramatically [6, 7, 8, 9].

Along with other microarchitectural advances in power-efficient performance, multicore processors have addressed these issues, successfully improving upon their single-core counterparts [10]. Current multicore designs integrate multiple, identical cores per chip, running at relatively low clock speeds to provide high performance at low power, with favorable thermal characteristics. Even though increasing the number of execution cores per processor has become the most straightforward means of increasing hardware performance, a new trend of diminishing returns [3] is beginning to become apparent when looking ahead to the future of symmetric multicore designs. This stems from the fact that increases in performance cannot ultimately keep pace with increases in the speed and number of transistors per processor core.

On the one hand, as process technology scales downward (from 65nm to 45nm, for example, if we look at Intel processors' road map), transistors become faster, and more of them can fit per unit of surface area. On the other hand, the performance of some microarchitectural structures, such as cache memories, increases at a less-than-linear rate as the components become smaller [3]. Hence, on-die technology scaling will increasingly become a limiting factor to processor performance.

In the quest to create more power efficient CPUs, several researchers proposed asymmetric multicore designs, based on the integration of non-uniform cores with different functional specialties and capability levels. These asymmetric designs potentially allow more performance to be packed into a given amount of space, while saving a significant amount of power over conventional symmetric multicore processors. For these reasons, asymmetric multicore computing is a likely candidate to be the next set of changes on this order of magnitude. This transition will extend the

capability of multicore processors to build performance and efficiency, giving rise to a significant set of software challenges and opportunities.

The terms *asymmetric* and *heterogeneous* are commonly used synonymously to refer to this type of multicore architecture. However, *asymmetric* tends to be favored more in discussions of software (since it implies the manner in which the hardware architecture is exposed to software), whereas *heterogeneous* tends to be favored more in discussions of hardware (since it more directly describes the physical architecture itself). For the sake of simplicity, in this thesis we have used the term "asymmetric" throughout.

1.1. Asymmetric multicore processors

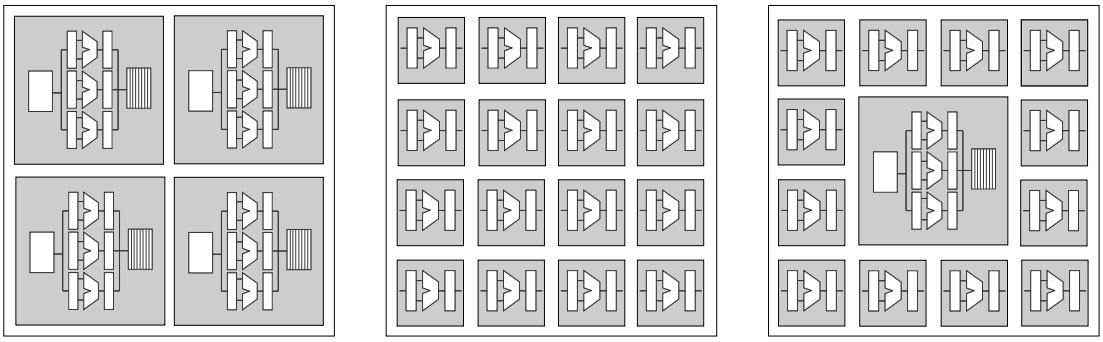
Most general-purpose multicore processors consist of identical cores: either the large, complex and powerful ones, as in the Intel Xeon or AMD Opteron processors, or the small, simple and lower-power ones, as in Sun's Niagara [11]. Cores in the former approach, implement sophisticated microarchitecture characteristics, such as superscalar and out-of-order execution, to achieve high single-thread performance. Previous research has shown that this approach can incur high energy costs as the number of cores continues to grow [3]. The latter approach, however, sacrifices single-thread performance at the expense of delivering benefits to applications with thread-level parallelism.

In terms of providing the optimal performance/area ratio, current *symmetric* multicore processors are ideally suited for some applications but not all. Figures 1.1a and 1.1b depict two hypothetical area-equivalent symmetric multicore chips consisting of a few complex cores and numerous simple cores, respectively. Suppose, for example, that any complex core delivers roughly twice the single-threaded performance of simple ones[1]. According to this information, the processor consisting of complex cores would run a sequential application twice as fast as the simple-core-based one. Conversely, the multicore processor integrating sixteen simple cores turns out to be a better choice for running a scalable parallel application, since it would obtain a 2× wall clock speedup with respect to executing this kind of application on the processor with four complex cores[2]. This example highlights that specific symmetric multicore designs are suitable for some applications, but fail to deliver significant benefits across a broad spectrum of applications.

To overcome this limitation, hardware designers proposed *asymmetric multicore processors* (AMPs) [1, 3]. A typical asymmetric multicore processor (as shown in Figure 1.1c) consists of a few fast and powerful cores (high clock frequency, complex out-of-order pipeline, and high power consumption) and a large number of slower

[1] The numbers to estimate the conversion ratios of performance and area in complex vs. simple cores were obtained from Hill and Marty [5]

[2] This estimate can be easily obtained by applying Amdahl's Law [12], provided that the parallel application has a negligible sequential fraction and it is allowed to use all cores available in the platform.



(a) A symmetric multicore processor consisting of 4 complex cores

(b) A symmetric multicore processor consisting of 16 simple cores

(c) An asymmetric multicore processor including 12 simple cores and 1 complex core

Figure 1.1: Three hypothetical area-equivalent multicore processors. Note that these figures illustrate area, not power, as the chip's limiting resource and omit important structures such as memory interfaces, shared caches, and interconnects.

and simpler cores (low clock frequency, simple pipeline, and low power consumption) all exposing the same ISA (Instruction Set Architecture). Integrating different types of cores in the same processor enables asymmetric designs to deliver the best of both worlds: scalability and power efficiency for parallel applications and fast single-threaded performance for sequential applications. Previous studies demonstrated that AMPs can achieve up to 63% better performance than symmetric multicore processors of comparable power and area budget [3].

Apart from delivering benefits to a wide range of applications with a common processor design, AMP systems also promise significant reductions in power consumption over their symmetric counterparts [1, 4, 13]. For example, because of performance/power trade-offs between complex and simple cores, it turns out to be much more efficient to run a parallel application on a large number of simple cores than on a smaller number of complex cores that consume the same power or fit into the same area [5, 14, 4]. In a similar vein, complex and powerful cores are suitable for running CPU-intensive applications that effectively use those processors' advanced microarchitectural features, such as out-of-order super-scalar pipelines, advanced branch prediction facilities, and replicated functional units [1, 2]. At the same time, simple and slow cores deliver a better trade-off between energy consumption and performance for memory-intensive applications that spend a majority of their execution time fetching data from off-chip memory and stalling the processor [15].

1.2. Design space of asymmetric multicore architectures

For the sake of clarity, the discussion above simplifies the potential differences between the various cores within the processor package, characterizing them as large, complex and power-hungry, versus small, simple and power-efficient. That sort of

asymmetry might be underpinned by a number of hardware characteristics, ranging from clock speed and cache size to different functional capabilities.

In [16], the authors classify asymmetric architectures into two categories: *performance asymmetric* and *functional asymmetric*.

The former refers to architectures where all cores support the same instruction set architecture[3] (ISA), but differ in performance (and power) due to different clock speeds, cache sizes, microarchitectures, and so forth. In this type of architectures, also known as *asymmetric single-ISA*, the same application binary runs "correctly" on any core due to the common ISA among cores, which greatly simplifies software compatibility and development. Although systems with explicit performance asymmetry are not yet being built, much of the literature investigating the properties of these systems has been originated from major hardware manufacturers such as Intel [4, 17, 8] and HP [1, 3, 13], indicating that within the industry there is interest in this architecture. Currently, it is possible to turn off-the-self symmetric multicore processors into asymmetric ones by configuring some of the cores to run at a lower-than-maximum voltage and frequency levels[4].

Functional-asymmetric architectures, on the other hand, encompass processors including cores with non-identical ISAs. Different types of general-purpose cores may be present with varying support for specific instruction sets. For example, it may be advantageous in terms of hardware complexity and silicon area that a processor supports vector instructions only on a subset of cores. Other types of functional asymmetry may range from different architectural registers or addressing modes to different memory hierarchy or exception and interrupt handling. In the extreme case, a processor may contain cores with fully disjoint ISAs, such as the Intel IXP [18] processors as well as some implementations combining homogeneous cores with accelerators (GPUs or even FPGAs [19]).

As far as application development is concerned, functional-asymmetric hardware usually brings more significant challenges than performance asymmetric one. For example, when functional asymmetry is present, programs compiled for one ISA may fail on cores with a different ISA, even when the difference is small. In consequence, software must handle these functional differences among cores appropriately in order to avoid runtime failures. The IBM Cell BE processor, is an example of a functional-asymmetric processor including cores which differ in most aspects of the ISA, but share the same data types and virtual memory architecture [20]. Despite the continuous progress made by compiler technology over the last few years, mapping applications onto these architectures is still extremely labor intensive. In most cases, developers must explicitly partition programs into different kernels, which are compiled to run on a particular core type. Then, either the application developer or the underlying runtime system maps and schedules these kernels onto the target system [21].

[3]The term ISA refers to the portion of a core that is visible to software, including instructions, architectural registers, data types, addressing modes, memory architecture, exception and interrupt handling, and external I/O.

[4] Dynamic voltage and frequency scaling (DVFS) facilities available on most modern CPUs allow to reduce the amount of power consumed by the processor.

Recent literature from main processor manufacturers suggests that next-generation asymmetric systems will expose both performance and functional asymmetry [16, 8]. Nevertheless, these future designs are likely to integrate cores that are identical in almost every aspect of the ISA except in a small set of instructions and architectural registers; designs where all cores share a physical address space with coherent caches. There are several reasons to believe that those specific asymmetric designs are firm candidates to be the next generation of multicore architectures. First, the fact that all cores support a large set of common instructions and a coherent address space greatly simplifies software compatibility. Second, designers will likely choose a base ISA for all cores and extend some cores with special features, such as wider vector processing, or diminish the capabilities of other cores to save power. Third, since processors from the same manufacturer usually support a large set of common instructions already, this model fits naturally and allows companies to reuse products at lower costs.

Wrapping up the discussion on the design space of asymmetric multicore architectures, it is worth noting that early studies have demonstrated that having just two core types is sufficient to extract most benefits from asymmetric systems [1]. Furthermore, having just two core types simplifies system design considerably. Therefore, it is highly likely that next-generation asymmetric systems will integrate two core types only: "fast" and "slow". Nevertheless, we expect future designs to have a variety of fast-slow core performance ratios, possibly targeting different market segments.

1.3. Thesis contributions

Asymmetric multicore systems enable us to employ *specialization*, namely, we can use each type of core for the type of computation where it delivers the best performance/energy trade-off. Previous research has demonstrated that different core specialization techniques contribute to improve efficiency of asymmetric multicore systems in diverse scenarios [1, 3, 2, 13, 5]. Unfortunately, specialization will not be delivered by the hardware, but it is up to the operating system (OS) and runtime system to unleash the potential of asymmetric multicore processors.

The main focus of this thesis is the design of OS-level scheduling algorithms for performance-asymmetric single-ISA multicore processors (AMPs from now on for brevity) with a two-fold objective. First, they must maximize system-wide performance on AMPs by devoting complex cores to running those threads in the workload that utilize them most efficiently. Second, the algorithms should deliver the potential benefits of these upcoming architectures to unmodified applications. Three novel scheduling algorithms are proposed in this thesis to fulfill those requirements: HASS, PA and CAMP.

The **Heterogeneity-Aware Signature-Supported (HASS)** scheduler attempts to maximize system-wide performance on AMPs by catering to the microarchitectural diversity of the workload. HASS is based on the idea of *architectural signature*,

a compact summary of microarchitectural properties of the application (e.g, memory access patterns or instruction-level parallelism). The information contained in an application's signature enables the scheduler to efficiently determine the relative benefit that it would derive from running on a complex, fast core over a simple, slow one. The first version of HASS (static HASS or HASS-S) was proposed in our previous work led by D. Shelepov [22]. In this thesis, we propose a more versatile dynamic version of this scheduler (HASS-D), which relies on online-generated architectural signatures rather than on offline-generated ones. Through an extensive evaluation of HASS-S and HASS-D, we demonstrate that these algorithms successfully overcome the main limitations of previously proposed schemes, which rely on different techniques to determine fast-to-slow relative speedups.

The **Parallelism-Aware (PA)** scheduler is the first OS-level scheduling algorithm specifically designed to automatically accelerate sequential phases of parallel applications on AMPs. To make that possible, PA ensures that sequential phases run on complex cores while the execution of parallel phases is relegated to simpler cores. PA is equipped with a sequential-phase-detection engine that enables it to effectively detect serial bottlenecks of parallel applications where unused threads block when waiting at synchronization primitives. In the event threads in the application use *spinning* for waiting (busy-wait), PA relies on a set of runtime extensions enabling the user-level threading library to notify the scheduler when a thread spins rather than doing useful work.

Finally, the **Comprehensive scheduler for Asymmetric Multicore Processors (CAMP)** addresses the main restriction posed by previously proposed algorithms: they are effective only for certain workload scenarios. This stems from the fact that they only cater to the microarchitectural diversity of the workloads or to the diversity in thread-level parallelism, but not both. CAMP exploits information on applications' TLP and ILP to make a comprehensive scheduling solution. Not only does CAMP deliver significant benefits for a wider range of workloads, but it is also able to outperform previous schedulers by leveraging knowledge on applications' TLP and ILP when making thread-to-core assignments.

The contributions of this thesis can be summarized as follows:

- The scheduling algorithms proposed in this thesis have been implemented on an actual operating system and extensively evaluated on real multicore hardware made asymmetric via dynamic voltage and frequency scaling (DVFS). Notably, none of these algorithms require changes to applications but only moderate changes to the target operating system, providing proof of concept towards real-world OS support for asymmetric hardware. Focusing our study on real implementations of these scheduling strategies led us to discovering interesting insights that would have been impossible to detect via simulation. As a result, this thesis sheds light on the main challenges that system software developers will likely face while trying to maximize exploitation of AMPs.

- Previously-proposed asymmetry-aware schedulers determined the speedup that a thread experience on a complex core relative to a simpler one, by means of

performance measurements on *both* core types [3, 2]. We found that such a monitoring methodology manifested serious performance problems, making it difficult to use it in practice. Bringing to light the problems with seemingly simple and effective monitoring methodologies and proposing asymmetry-aware algorithms that are not susceptible to similar deficiencies are key contributions of this thesis.

- The capability of asymmetric multicore systems to mitigate sequential bottlenecks of parallel applications has been widely demonstrated through analytical studies and via limited user-level scheduling prototypes [4, 5, 23]. In this thesis we designed and implemented the first OS-level scheduling algorithm delivering this potential of asymmetric systems to unmodified applications. By means of our study, we were able to identify the main barriers to realizing this potential at the OS level and determine to what extent the operating system alone can detect serial phases of parallel applications. Moreover, we demonstrate that the interaction between the runtime system and the OS is paramount to take full advantage of this outstanding capability of asymmetric systems.

- Theoretical and simulation-based studies concluded that both the instruction-level parallelism (ILP) and the thread-level parallelism (TLP) of an application contribute to the efficiency the application derives from using complex cores over simple ones [1, 5]. In this thesis, we derived a novel metric, the *Utility Factor* (UF), which accounts for both the ILP and TLP of the application and produces a single value that approximates how much the application as a whole will improve its performance if its threads are allowed to occupy all the complex cores available on the AMP. Such a metric is the foundation of our CAMP scheduler.

We must highlight at this point that our algorithms have been specifically designed to cope with performance asymmetry rather than with functional asymmetry, which, as stated previously, may be present in a small extent among cores in future systems (identical cores in almost every aspect of the ISA but in a small subset of it). Nevertheless, the proposed algorithms can be augmented with runtime techniques aimed to avoid runtime failures stemming from the presence of functional differences among cores, such as the *fault-and-migrate* algorithm presented in [16].

1.4. Thesis structure

The remainder of this thesis is organized as follows:

- Chapter 2 introduces the main *core specialization* techniques proposed to date for AMPs and analyzes the main barriers to exploiting these techniques. We also discuss how current general-purpose operating systems can be re-designed

to deal with next-generation asymmetric hardware, and describe our approach to designing asymmetry-aware scheduling algorithms, carefully crafted to take full advantage of specialization.

- Chapter 3 presents key aspects of our experimental test bed, such as the operating system we used to implement our scheduling algorithms as well as the hardware platforms where we carried out our evaluation. Given that asymmetric multicore systems are not available today, we also provide an overview of the different techniques available for the creation of experimental test beds for advance work in this area.

- Chapter 4 focuses on scheduling algorithms aimed to maximize system-wide performance on AMPs by catering to the microarchitectural diversity present in multiprogrammed workloads. Among the schedulers evaluated in this chapter, we propose HASS, which overcomes serious limitations that became apparent when evaluating real-world implementations of previously proposed algorithms.

- Chapter 5 illustrates the capability of asymmetric multicore systems to mitigate sequential bottlenecks of parallel applications. In this chapter, we demonstrate that this potential is realizable by means of the PA (Parallelism-Aware) scheduler, which caters to the amount of thread-level parallelism in the applications and wisely arbitrates the utilization of the "scarce" complex cores in an AMP.

- Chapter 6 presents our most relevant proposal: CAMP, a Comprehensive scheduler for Asymmetric Multicore Processors. The CAMP scheduler exploits the diversity in thread-level parallelism and microarchitectural diversity present in the workload to make efficient utilization of asymmetric multicore processors.

- Chapter 7 outlines the main conclusions of this thesis an discusses future work.

Chapter 2

Unleashing the Potential of Asymmetric Multicore Processors through OS scheduling

Asymmetric multicore systems, which integrate a mix of power-hungry complex cores and power-efficient slow cores, are very attractive because they can achieve high single-threaded performance and at the same time, deliver high performance thread-level parallelism with lower energy costs [1, 5]. Despite these benefits, AMPs give rise to significant challenges to the system software. Thread scheduling is one of the most critical challenges.

To fully tap into the potential of AMP systems, each core type must be *specialized* for applications that will use it most efficiently, thus ensuring the best performance/energy trade-off. To date, several core specialization techniques have been devised to improve efficiency of AMPs in diverse scenarios [2, 3, 4, 13]. Unfortunately, specialization will not be delivered by the hardware, but it is up to the system software to deliver the benefits of asymmetric systems to unmodified applications.

The fact of the matter is that conventional operating systems are not yet ready to take full advantage of specialization since they are not designed to cope with cores with different performance. For that reason, the OS must be redesigned to unleash the potential of AMPs. In particular, new scheduling policies should be devised to cater to the asymmetric properties of the system and to the applications's characteristics when performing thread-to-core assignments.

The aim of chapter is to shed light on these issues. It is structured as follows. In Section 2.1 we introduce the most relevant core specialization techniques proposed to date. Section 2.2 discusses the main challenges system software will have to face when attempting to unleash the potential of specialization. Section 2.3 presents our approach to redesigning current general-purpose operating systems so that they are able to make an efficient use of performance-asymmetric multicore architectures. Finally, in Section 2.4, we discuss additional challenges that operating systems

developers will likely encounter when designing asymmetry-aware schedulers which pursue additional goals, different from simply maximizing system-wide performance.

2.1. Specialization on AMPs

2.1.1. ILP specialization

In a multi-programmed computing environment, threads of execution exhibit different runtime characteristics and hardware-resource requirements. On AMP systems, these microarchitectural characteristics determine the relative speedup that each thread derives from running on a fast core rather than a slow one [1, 2, 3]. *ILP Specialization* caters to the diversity in threads' relative speedups present in multi-application scenarios.

Some programs are very efficient at using the CPU pipeline: they have a high instruction-level parallelism, meaning that a processor can issue many instructions in parallel without running out of work. These programs show good locality of memory accesses. As a result, they rarely access the main memory and thus rarely stall the processor. We refer to these programs as *CPU intensive*.

At the other extreme are programs that use the CPU pipeline very inefficiently. They typically have a high processor cache-miss rate and thus stall the CPU pipeline, because they have to wait while their data is being fetched from main memory. We refer to these programs as *memory intensive*[1].

CPU-intensive programs use the hardware of fast cores very efficiently; thus, they derive relatively large benefits from running on fast cores relative to slow cores. Memory-intensive applications, on the other hand, derive relatively little benefit from running on fast cores. Figure 2.1 shows the relative speedups of the SPEC CPU2006 benchmarks running on an emulated AMP system[2]. Note that some applications experience a 2× speedup, which is proportional to the difference in the CPU frequency between the two processors. These are the CPU-intensive applications that have a high utilization of the processor's pipeline functional units. Other applications experience only a fraction of the achievable speedup. These are the memory-intensive applications that often stall the CPU as they wait for data to arrive from the main memory, so increasing the frequency of the CPU does not directly translate into better performance for them. For example, the memory-intensive application `lbm` speeds up by only 33% when running on the fast core.

When running multi-application workloads, it is more profitable for system-wide efficiency to run CPU-intensive programs on fast cores and memory intensive pro-

[1] Note that this is not the same as an I/O bound application, which often relinquishes the CPU when it must perform device I/O. A memory-intensive application might run on the CPU 100% of its allotted time, but it would use the CPU inefficiently.

[2] A symmetric multicore system was used to emulate this AMP by using dynamic frequency scaling. Fast cores were emulated by setting their frequency at 2.3GHz while slow cores were emulated by using the frequency of 1.15GHz.

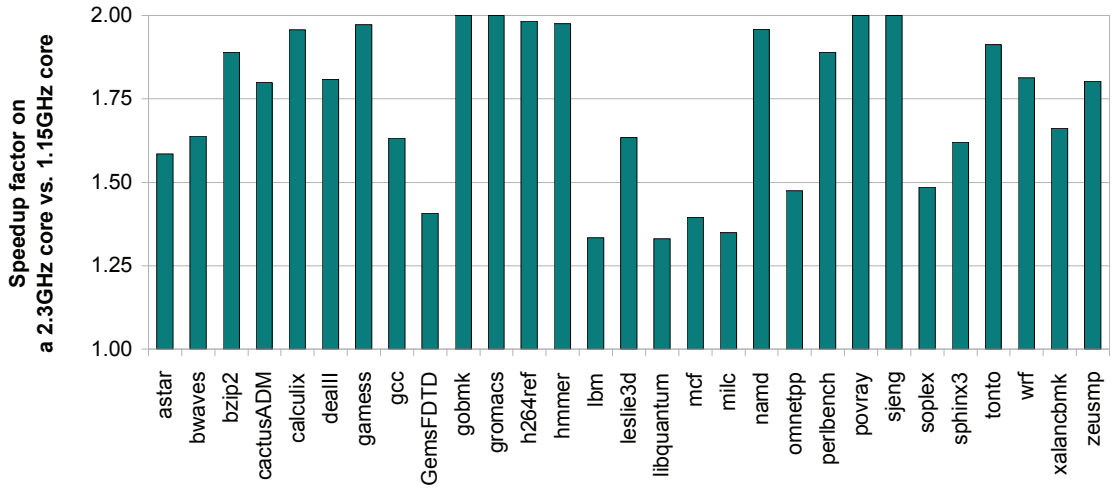


Figure 2.1: Relative speedup experienced by applications from the SPEC CPU2006 benchmark suite from running on a fast core (2.3GHz) vs. a slow core (1.15GHz) of an emulated AMP system. The maximum achievable speedup is a factor of 2. The more memory intensive the application is the less speedup it experiences.

grams on slow cores. This is what catering to microarchitectural diversity of the workload is all about. Recent work from the University of California, San Diego and HP demonstrated [1, 3] that AMP systems can offer up to 63% better performance than SMPs comparable in area and power, provided that the operating system employs a scheduling policy that caters to the microarchitectural diversity of the workload.

For the sake of simplicity, the discussion above states that applications with frequent pipeline stalls due to memory accesses make ineficient utilization of complex pipelines, and that results in a low fast-to-slow relative speedup. Although memory behavior is the only predictor required to determine relative speedups when cores have the same microarchitecture but differ in processor frecuency, other forms of asymmetry may force the system designer to consider further performance-limiting factors. For example, on AMPs where cores differ in retirement width, very frequent stalls originated internally to the processor package (such as those associated to ITLB misses or branch mispredictions) may lead to substantially reducing the fast-to-slow relative speedup even for CPU-intensive applications [24].

2.1.2. TLP specialization

TLP specialization caters to the diversity in thread-level parallelism present in the workload. This sort of diversity refers to two broad categories into which applications can be classified: *scalable* applications and *non-scalable* applications. Scalable parallel applications use multiple threads of execution, and increasing the number of threads typically leads to reduced execution time or increased amount of work performed per unit of time. The term *non-scalable* application, on the other hand, encompasses both purely sequential applications and parallel applications that scale

only up to a fixed number of threads. These applications use effectively only one or a small number of cores. In addition to purely parallel or purely sequential applications, there is a hybrid type, where an application might exhibit phases of highly parallel execution intermixed with sequential phases.

Multicore architectures is good news for scalable parallel applications, as with each new technology generation their performance will increase proportionally with the number of cores in the new processor. Unfortunately, the news is not so good for single-threaded applications or for parallel applications that scale only up to a fixed number of cores. These applications will receive no benefit from an increasingly large number of cores, and since the speed of individual cores will stay fairly flat, their performance will stay flat as well[3].

This is a serious problem for the software development community and even for the software industry itself. The fact of the matter is that not all applications are parallel, and not all parallel applications scale to a large number of cores. While there is a strong drive towards parallelization, many programs will remain sequential (or will have limited scalability) simply because parallelizing the software is very expensive[4], because some algorithms are inherently sequential, and because there will be legacy programs whose source code is unavailable.

Asymmetric multicore processors enable us to address this problem by providing different types of processing cores, well-suited to achieve the best trade-off in performance and energy consumption for both parallel and sequential applications. Suppose we have a scalable parallel application with a choice of running it on a processor either with a few complex and powerful cores or with many simple low-power cores. For example, suppose we have a processor with four complex and powerful cores and another area-equivalent and power-budget-equivalent processor consisting of 16 simple, low-power cores. Suppose further that each simple core delivers roughly half the performance of one complex core[5]. We configure the number of threads in the application to equal the number of cores, which is a standard practice for compute-intensive applications [26]. If we run this parallel application on the processor with complex cores, then each thread will run roughly twice as fast as the thread running on the processor with simple cores (assuming that threads are CPU-intensive and that synchronization and other overhead is negligible), but we can use only four threads on the complex-core processor vs. 16 threads on the simple-core processor. Since using additional threads results in a proportional performance improvement in this application, we get twice as much performance running on a simple-core processor as on a complex-core processor. Recalling that these two processors use the same power budget, we achieve twice as much performance per watt.

[3]Techniques for accelerating single-threaded applications on multicore processors such as speculative multithreading [25] have limitations.

[4]Tim Sweeney the designer of multi-threaded Unreal 3 video game engine, stated that "Implementing a multithreaded system requires two to three times the development and testing effort of implementing a comparable non-multithreaded system" [7].

[5]The numbers to estimate the conversion ratios of performance and power in complex vs. simple cores were obtained from Hill and Marty [5].

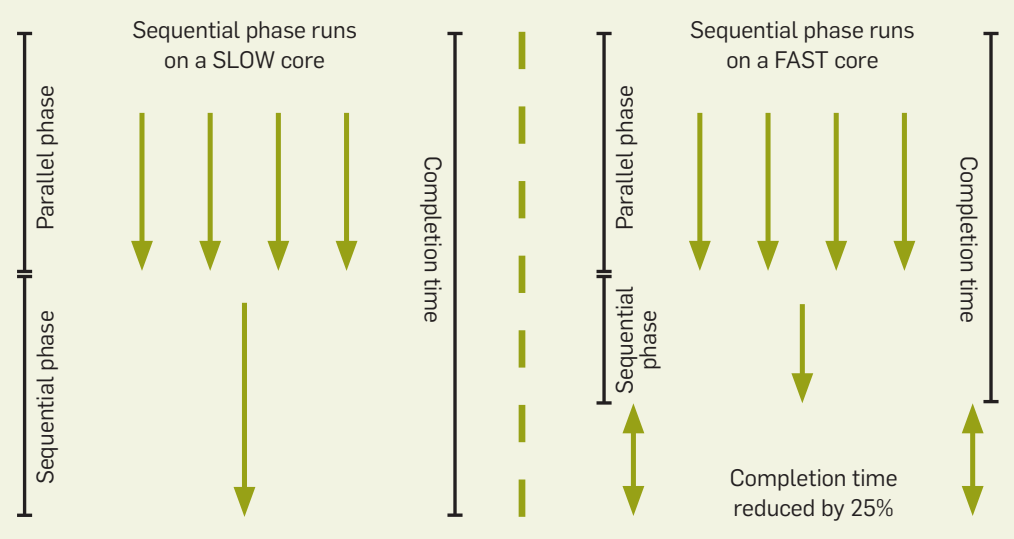


Figure 2.2: An illustration of how an scheduler delivering TLP specialization would accelerate a parallel application limited by a sequential bottleneck on an AMP.

Contrast this to running a sequential application, which cannot increase its performance by using additional threads. Therefore, using a single thread, it will run twice as slow on a simple-core processor than on a complex-core processor, meaning we get twice as much performance per watt running on the complex-core system. An experienced reader will observe that power consumption on a single-core system for the single-threaded application workload could be reduced by turning off unused cores. Unfortunately, it is not always possible to turn off unused cores completely, especially if they are located in the same power domain as the active cores. Furthermore, an operating-system power manager may be configured to avoid putting the unused cores in a deep sleep state, because bringing the cores up from this state takes time [27]. Thus, if a new application begins running or if the operating system needs a core for execution, then additional latency will be incurred while the dormant core is being brought up in the active power state. This example demonstrates that applications with different levels of parallelism require different types of cores to achieve the optimal performance-per-watt. AMP systems offer the potential to resolve this dilemma by providing cores of both types.

Another advantage of having both "fast" and "slow" cores in the system is that the fast ones can be used to accelerate sequential phases of non-scalable parallel applications, mitigating the effect of sequential bottlenecks and reducing effective serialization. For example, consider a parallel application with 50% of its code executing sequentially. Suppose further that the fast cores run twice as fast as the slow cores. In this scenario mapping the serial part of the application onto a fast core will result in a 25% performance improvement (see Figure 2.2). All in all, Hill and Marty demonstrated that for parallel applications with sequential phases AMPs can potentially offer significantly better performance than SMPs, as long as sequential phases constitute as least 5% of the execution time [5].

2.1.3. Other specialization techniques

A large body of work has been devoted to investigating scheduling algorithms exploiting *ILP specialization* and *TLP specialization*. Despite of the fact that additional core specialization techniques have been proposed, these are related, one way or another, to the two "basic" forms of specialization analyzed in this thesis.

Of special attention among these additional techniques is the utilization of AMPs to save energy on operating systems. Mogul et al., first proposed that the idea behind ILP specialization (mappping threads with low relative speedups to simple cores) could be extended to OS code [13]. Essentially, this can be effected by devoting low-power cores to the execution of operating system code, as well as by powering down complex CPU cores when idle.

The motivation for this specialization technique stems from several facts. First, OS code does not proportionately benefit from the potential speedup of complex, high-frequency cores[6]. As a result, OS code execution on complex CPU cores wastes energy, because it does not fully exploit that complexity. Conversely, running OS code on simpler cores leads to a better use of power and chip area[7]. Second, it is well known that many applications spend much of their execution in operating system code [29, 30]. Those OS-intensive applications lend themselves to energy-aware optimizations that rely on mapping on simple, power-efficient cores those parts of the application spent on the OS code [13]. Third, most computer systems (with exceptions, such as those used for scientific computing) are often idle[8]. Therefore, powering down complex CPU cores when they would otherwise be idle improves power efficiency.

Along with the appropriate OS support, asymmetric multicore systems enable us to restore energy proportionality to OS code execution. This can be accomplished by powering down high-power high-complexity "application" cores, while maintaining OS functions on low-power low-complexity "OS-friendly" cores. For example, the arrival of a new Web (HTTP / TCP) connection normally precedes the arrival of the actual HTTP request by at least one network round-trip time (typically on the order of milliseconds or more). This delay would allow an OS core to handle the initial TCP connection request, letting the application core (if otherwise idle) continue sleeping until the actual HTTP request arrives.

Overall, this specialization technique could be applied to OS code in several ways:

- *Dynamically switching between cores in OS kernel code.* This technique consists in moving a thread of execution from one core to another when executing certain system calls.

[6] In 1990, John Ousterhout observed that "operating system performance does not seem to be improving at the same rate as the base speed of the underlying hardware" [28].

[7] Nellans et al. observed that "a classic 5-stage pipeline" such as a 486 is surprisingly close in performance to a modern Pentium 4 when executing OS code [29].

[8] Ranganathan et al. report 90th-percentile utilization from nine different enterprise clusters between 5.3% and 44% [31]. A subset of Google servers operates "most of the time" between 10% and 50% utilization [32].

- *Binding device interrupts to OS-friendly cores.* Network interface controllers (NICs), which interrupt more often than most other devices, could take advantage of this technique. Note that the Linux kernel already offers this functionality.

- *Binding kernel threads to OS-friendly cores.* Grant and Afsahi have shown that this improves energy efficiency on multithreaded scientific applications [33].

- *Running virtualization "helper processing" on low-power cores.* Systems such as Xen [34] use a privileged virtual machine ("Domain") to multiplex and manage I/O operations for other virtual machines. This code could run on a low-power core.

2.2. From specialization to scheduling

Specialization techniques enable to maximize the overall system performance, and at the same time, deliver a better performance per watt and per area. Unfortunately, specialization will not be delivered by the hardware, but it is up to the system software to employ asymmetry-aware scheduling policies that tailor asymmetric cores to the instruction streams that use them most efficiently.

Here, we discuss the three main challenges involved in delivering the benefits of the ILP specialization and TLP specialization to unmodified applications: (1) determining applications' relative speedups, (2) detecting sequential and parallel phases and (3) reducing the overhead stemming from cross-core migrations.

2.2.1. Determining relative speedups

Recall that the idea behind ILP specialization is to assign threads (or phases of execution) with high relative speedup to fast cores and threads (or phases) with low relative speedup to slow cores. For example, CPU-intensive code will experience a higher relative speedup running on fast vs. slow cores than memory-intensive code (as shown in Figure 2.1), so scheduling it on fast cores is more efficient in a cost-benefit analysis.

The biggest challenge in implementing an algorithm that caters to ILP specialization is to determine running threads' relative speedups as they go though different phases of execution at scheduling time. Two approaches were proposed in the research community to address this problem.

The first approach entails running each thread on cores of different types, registering the speedup obtained on a fast core relative to a slow core and using the resulting relative speedup to drive thread-to-core assignments. In a scheduling algorithm, a thread with a larger relative speedup would be given preference to run on a fast core, and a thread with a lower relative speedup would be more likely to run on a

slow core. Since this approach relies on direct measurement of relative speedup, we refer to it as the *direct measurement* approach.

A second approach, referred to as the *modeling* approach, is to model an application's speedup on a fast vs. slow core using its runtime properties obtained either offline or online. Modeling is less accurate than direct measurement but does not require running each thread on each type of core. Hence, it avoids serious problems stemming from direct measurement.

In an effort to build asymmetry-aware algorithms that cater to microarchitectural diversity, we have experimented with both methods in this thesis. We found that direct measurement approach manifested several performance problems. Consider a scenario where each thread must be run on each core type to determine its relative speedup. Given that a running thread may switch phases of execution (that is, it may be doing different types of processing at different points in time), this measurement must be repeated periodically; otherwise, the scheduler might be operating on stale data. Since the number of threads will typically be larger than the number of fast cores, there will always be a high demand for running on fast cores for the purpose of remeasuring relative speedup. As a result, threads that are "legitimately" assigned to run on fast cores by the scheduling policy will observe undue interference from threads trying to measure their speedup there. Furthermore, having too many threads "wishing" to run on scarce fast cores may cause load imbalance, with fast cores being busy and slow cores being idle. When we used this direct measurement approach in an asymmetry-aware algorithm, we found that these problems made it difficult to deliver significant performance improvement relative to an asymmetry-agnostic scheduler. We will elaborate on these issues in Chapter 4.

The modeling approach, on the other hand, involves predicting relative speedup on different core types using certain properties of the running programs. Although this approach overcomes the shortcomings of direct measurements, it requires to be equipped with an accurate estimation model for the asymmetric platform in question. With such an accurate model, the asymmetric aware scheduler can effectively exploit the microarchitectural diversity of the workload. All in all, designing accurate prediction models on *highly asymmetric* systems, where not only do fast and slow cores differ in processor pipeline and frequency but also differ in cache sizes, can be a challenging task. In these scenarios, estimations based on memory behavior can be used as a first approximation since memory access will likely remain as the major performance-limiting factor in asymmetric systems (as recently found in [24]).

2.2.2. Detecting sequential and parallel phases

In order to fully tap into the potential of TLP specialization, the thread scheduler must be able to detect parallel and sequential phases in applications. The simplest way to do this is to use the runnable thread count as a heuristic. If an application uses a large number of threads, then its runnable thread count will be high, and

this application would be in a parallel phase. Conversely, an application with a single runnable thread would be in a sequential phase.

A good property of the runnable thread count is that it is visible to the operating system in modern multithreading environments, because these systems perform a one-to-one mapping between application-level and kernel-level threads. Therefore, by monitoring the runnable thread count, the operating system can distinguish between parallel and sequential phases in applications.

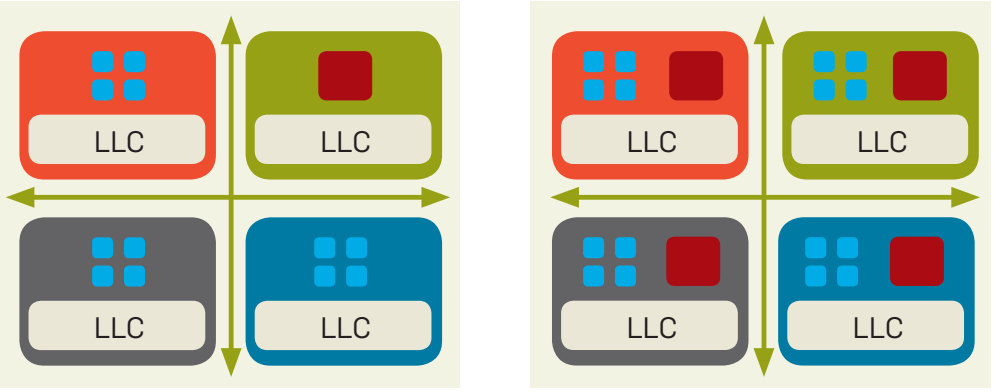
Unfortunately, in some cases, using the runnable thread count for detection of sequential phases may not work. In particular, an application could be running non-scalable code while still using a large number of runnable threads. We describe two scenarios where this might happen and discuss potential remedies.

In the first scenario, an application might be susceptible to an external scalability bottlenecks –for example, as a result of memory bandwidth contention. In this case, the system memory bus may be saturated, and using additional threads does not speed-up the application because those threads do not contribute to useful computation. A sensible way to solve this problem is to reduce the number of threads used in an application to the point where the application operates at its peak efficiency. Essentially, this boils down to configuring the number of threads properly in a parallel application. Suleman et al. describe a technique called *feedback-driven threading*, which enables to dynamically determine the optimal thread count for parallel applications [35].

In the second scenario, an application might be limited by internal scalability bottlenecks: for example, there might be a load imbalanced situation where some threads do more work than others, or serial bottlenecks where one thread executes code inside a critical section while other threads wait. When threads wait they may either block, relinquishing the CPU, or busy-wait, spinning on the CPU in a tight loop. The choice of waiting mode has a strong impact on the performance and scalability of these applications: spinning provides maximum performance but wastes significant processor resources, while blocking-based approaches conserve processor resources but introduce high overheads stemming from an increased number of context switches. Note, however, that purely blocking and purely spinning synchronization modes are seldom used[9]. Instead, most synchronization libraries implement two-phase (a.k.a. adaptive) synchronization algorithms where threads busy-wait for a while and then block (a settable *spin threshold* usually defines the maximum amount of spinning), avoiding this way costly context switches when waiting times are short [36].

The detection of serial phases by simply examining the application's runnable thread count would work well as long as unused threads block during serial phases. However, this heuristic fails in situations where unused threads perform busy waiting (spinning) and remain runnable even though they do not make useful progress. In

[9]Purely blocking approaches are beneficial when the number of threads is greater than the number of cores, since threads doing useful work do not have to share the CPU with spinning threads. Conversely, busy-waiting makes sense on a multiprocessor when the wait times are expected to be very short.



(a) (b)

Figure 2.3: Two configurations of AMP systems. Large squares represent memory domains with one or more cores and a last-level cache (LLC) inside the domain. "Fast" cores are denoted by large red boxes, "slow" cores are denoted by small blue boxes.

this case, we have a problem of *hidden serial phases*. Adaptive synchronization diminishes the problem of hidden serial phases, but does not eliminate it entirely, because it still requires that threads spin. Although these spinning periods are relatively short, frequent spinning on fast cores is a waste of processor and energy resources.

In this thesis, we propose a set of optimizations to address this problem by enabling our scheduler to interact with the threading library or runtime environment (see *PA Runtime Extensions* in Section 5.1.2). With these extensions, the runtime system can notify the scheduler when a thread spins rather than doing useful work and also specify which thread is most likely to execute serial code. We found that this interaction enables asymmetric-aware schedulers to accelerate hidden serial phases and improve the performance of applications. Furthermore, these optimizations do not require any changes on parallel applications themselves since notifications are performed by the underlying threading library.

2.2.3. Mitigating migration overhead

Another challenge in implementing asymmetry-aware algorithms is to mitigate the overhead associated with migrating threads across the cores. Overall, any asymmetry-aware algorithm catering to core specialization techniques relies on cross-core migrations to deliver the benefits of its policy. For example, an algorithm exploiting TLP specialization must migrate a thread from a slow core to a fast core if it detects that the thread is executing a sequential phase. Likewise, an algorithm taking advantage of the microarchitectural diversity present in the workload, may require to readjust the thread-to-core assignments via thread migrations when detecting transitions between CPU-intensive and memory-intensive phases of the applications.

Migrations are an essential tool of asymmetry-aware algorithms, but unfortunately

they can be quite expensive. The asymmetric multicore system shown in Figure 2.3a consists of several memory domains, as is usually the case with modern multicore processors. A memory domain is defined to contain cores that share an LLC (last-level cache). LLC is the last "line of defense" on the frontier between the CPU and the main memory. Thus, if the required data is not in the LLC, then the processor has to fetch it from the main memory, which takes hundreds of CPU cycles and may slow down the computation considerably. In contrast, fetching data from an LLC takes only a few tens of processor cycles.

Therefore, it is desirable to minimize the number of accesses to the main memory and try to satisfy data requests from an LLC or other CPU caches as frequently as possible. In Figure 2.3a, the fast core is located in a different memory domain from the slow cores, so every time the scheduler migrates a thread to the fast core, the thread loses the data accumulated in the LLC of the slow core's memory domain and must fetch the data from the main memory[10]. In this platform, migrations can cause significant performance overhead.

Consider now Figure 2.3b, which depicts a different AMP system where each fast core is located in the same memory domain as several slow cores. This "migration-friendly" architecture makes it easier for the thread scheduler to avoid cross-memory-domain migrations. In this case, a scheduler will try to migrate a thread to a fast core that is within the same memory domain as the slow core where the thread was previously running, thus enabling the thread to reuse the data in the LLC.

In order to leverage knowledge on migration overhead, the scheduler must be aware of the topology of the asymmetric system. Such *topology-aware* design enables the scheduler to avoid cross-memory-domain migrations whenever is possible, effectively mitigating migration overhead. In this thesis, we demonstrate the potential benefits of "migration-friendly" asymmetric multicore architectures properly equipped with a topology-aware thread scheduler (see Section 5.2.4).

2.3. Rethinking the scheduling subsystem

Conventional operating systems, such as all variants of Linux and Solaris, are not yet ready to deal with systems including cores with different performance. Not only does the asymmetry unawareness of the OS results in an ineffective utilization of the platform –since core specialization techniques are not exploited–, but also causes unpredictable behavior. This stems from the fact that, without considering hardware asymmetry, the *default* scheduler in these systems may map a thread to a higher performance core in one run, but to a lower performance core in another run. As a result, the fraction of time that a thread spends on a particular core type may vary significantly from one run to another, giving rise to non-repeatable performance results.

[10] Depending on the implementation of the processor, the thread might have to fetch the data from its old LLC, not from memory, but that is still more expensive than fetching it from the LLC in its current memory domain.

Yet, it is worth bearing in mind that symmetric multicore systems will likely coexist with asymmetric ones in the near future. For this reason, the schedulers of general-purpose operating systems must be properly designed to be able to cope with both symmetric and asymmetric multicore designs, ensuring an effective utilization of these systems.

In a similar vein, there are several scenarios where it might turn out beneficial to set up an asymmetric system to work with cores of only one type at a time, thus operating as a symmetric system. For example, power-hungry cores can be temporarily disabled to save energy on platforms with strong power constraints (e.g., mobile devices) or when a highly scalable parallel application runs alone on the system (it can execute very efficiently by spreading its computation across abundant low-power cores). Conversely, if our workload consists of just a few CPU-bound sequential applications, then the performance of the system can be maximized –at the cost of high power consumption– by using as many complex cores as needed and potentially turning off all low-power cores as well as unused power-hungry ones.

In this thesis, we have modified the Solaris kernel to demonstrate that, with moderate changes, an existing OS can be extended to effectively support both symmetric and asymmetric-performance single-ISA hardware with a common scheduler design. Only a modest set of changes was necessary to incorporate the basic support into the core kernel (as we will show in Sections 3.4.2 and 3.4.4). Notably, a substantial amount of new source code was devoted to the creation of scheduling modules specifically designed for asymmetric multicore hardware.

The aim of this section is to describe our approach to re-designing a general-purpose operating system so that it can make efficient use of performance-asymmetric multicore architectures. For the sake of generality, this section only provides an overview of the design of our scheduling framework, which includes high-level aspects not tied to any specific operating system kernel. Thus, it can serve as a design guide for implementing asymmetry-aware schedulers in any target operating system. It is worth noting, however, that further information about the implementation of our scheduling algorithms in the Solaris kernel can be found in Section 3.4.

The remainder of this section is organized as follows. Section 2.3.1 focuses on how to make the operating system aware of the presence of cores of different types. In Section 2.3.2, we analyze the problem of load balancing on asymmetric multicore systems and describe our approach to load balancing and thread-to-core assignments on AMPs. In Section 2.3.3, we introduce the different modes of operation of our scheduling subsystem, which make it possible for the OS to deal with symmetric and performance-asymmetric systems with a common design.

2.3.1. Detection of different core types

To detect the different features present in current multicore processors, modern operating systems rely on certain assembly instructions specifically included in the ISA for that purpose. For example, on x86 processors, the `cpuid` assembly instruction [37] allows the OS to discover key features of the CPU, ranging from the

supported instruction types on the different cores –such as the availability of MMX or SSE multimedia instructions; to the different properties of each level of the cache hierarchy –such as total size, line width or associativity. Other information such as the core frequency, can be obtained via the `rdmsr` assembly instruction, which enables the OS to read x86 MSRs (Model-Specific Registers) [38]. Operating systems for future asymmetric multicore hardware can use these special instructions to spot the differences across cores and effectively expose the different core types in an AMP to the thread scheduler.

Notably, some characteristics of the cores on current multicore hardware, such as the processor frequency, may change dynamically due to the action of hardware- or software- driven power-management mechanisms supported by modern processors [39]. For example, in an attempt to substantially reduce the amount of consumed power of the platform, the OS may decide to lower the DVFS level of some cores, which also results in a lower processor frequency and performance [27]. These dynamic changes in performance may cause trouble to the thread scheduler on AMPs, since the differences in performance between core types becomes dynamic as well. As a result, power management and asymmetry-aware scheduling must be performed cooperatively and in a fully coordinated fashion.

2.3.2. Load balancing and thread-to-core assignments

Most existing multiprocessor OSs for current multicore systems, such as Linux 2.6 series , Windows Server 2003, Solaris 10 and FreeBSD 5.2, are based on a distributed model, where the scheduler maintains per-core run queues to keep track of runnable threads in the system. With this model, load balancing is triggered periodically on every core in an independent fashion[11].

Existing load balancers of modern OSs define the *load of a core* to be the length of its run queue (number of threads assigned to it), and attempt to keep all run queues as evenly loaded as possible. When threads transition between runnable and non-runnable states, some run queues may become more loaded than others. In this scenario, thread migrations are required to enforce load balance. Current OSs' load balancers try to mitigate the negative performance effects that come from migrations by selecting to migrate those threads which are most likely to experience a low performance degradation after being migrated. For example, threads that have not run for a while on the core they are assigned to are suitable candidates for migration, since it is likely that little of their data remains in the cache hierarchy at that point.

Regrettably, this approach to load balancing does not work well as such on asymmetric hardware, since nothing prevents the scheduler from migrating threads between different core types at infrequent and arbitrary intervals. The fact of the matter is

[11] This model stands in contrast with a centralized run queue model, that uses a global run queue for all runnable threads. The distributed model tends to be more favored in modern implementations than the centralized one since it achieves better scalability.

that the fraction of time that a thread spends on the different core types may vary significantly across executions.

A potential way to address this problem is to make the load balancer aware of the computing power of the different core types in the platform. Li et al proposed in [17] an asymmetry-aware load balancing scheme based on this idea, which was the foundation for their AMPS scheduler. AMPS drives its load balancing decisions based on a new definition of *load of a core*, where the core's computing power is factored in. Our approach to asymmetry-aware load balancing is somewhat inspired by Li's approach. In an attempt to clarify the differences between our approach and Li's, we describe both solutions in terms of the *scaled computing power* and the *scaled load of a core*[12].

The traditional method to quantify CPU computing power is to run CPU-bound benchmarks, such as those in the SPEC CPU benchmark suite, and obtain metrics such as instructions-per-cycle (IPC) or million-instructions-per-second (MIPS). The OS can use this technique to quantify the computing power of the different core types in the asymmetric system when it boots for the first time. The *scaled computing power* of a given core, denoted by P, is defined as the core's computing power divided by the system's minimum core computing power. In [17], the authors emulate asymmetric systems including cores with different frequencies (just like we do in this thesis) and approximate scaled computing power using core frequencies rather than benchmark measurements. To make that possible, they set the scaled computing power of the slowest core in the platform to one and use $F \times S$ for the remaining cores, where F denotes the ratio between its frequency and system's slowest core frequency, and S is a less-than-one scaling factor that can be determined empirically. S reflects the fact that an F times higher frequency often leads to less than F times higher application performance since the memory system remains non-scaled.

Unfortunately, that approach poses two limitations: (1) it is specifically designed for cores that differ in processor frequency and (2) it does not directly cater to the diversity in the relative benefit that applications derive from running on a faster core rather than a slower one (as we showed in Section 2.1.1). To overcome these shortcomings, we opted to follow a slightly different and more general approach to determining the scaled computing power. To obtain this value, we ran all benchmarks in the SPEC CPU2006 suite (which cover a wide, representative range of program behaviors) on fast and slow cores and computed the relative wall clock time speedup that they experience on a fast core over a slow one. The scaled computation power for slow cores is set to one, while fast core's scaled computation power is set to the geometric mean of relative speedups obtained for the whole set of SPEC CPU2006 benchmarks[13]. All in all, our approach can be applied to any performance-asymmetric single-ISA system regardless of where the performance differences among cores in the platform come from, and, more importantly, it en-

[12]Both terms were introduced in Li's work for the first time.

[13]In our case, we opted to run the benchmarks with the reference input set, which may take a long time for the first time the OS boots. Nevertheless, the `train` input set can be used instead for this purpose, giving rise to a much more affordable first-time system boot.

ables the load balancer to account for the actual diversity in fast-to-slow speedups experienced by applications.

Li's AMPS scheduler performs load balancing in terms of the *scaled load* of the different cores. For any core with scaled computing power P, its *scaled load*, denoted by, L_c is defined as its run queue length divided by P. Let L_c^{max} and L_c^{min} be the maximum and minimum scaled load of a core in an asymmetric system. AMPS assumes that the system is load-balanced if $L_c^{max} - L_c^{min} \leq 1$. In addition to this novel load balancing scheme, AMPS follows a *faster-core-first-scheduling* approach –which ensures that fast cores never go idle before slow cores; and implements a fair-share mechanism extended to different core types. All these features enable AMPS to accomplish its main goals: repeatability of application performance and fairness.

The schedulers proposed in this thesis, however, attempt to maximize system-wide performance rather than fairness, which leads to dictating different algorithms for load balancing. More specifically, our approach to load balancing is strongly determined by the fact that our scheduling algorithms exploit core specialization techniques. As we showed in Section 2.1, these techniques allow us to identify those threads in the workload which utilize most efficiently the fast cores in the system. In essence, that makes it possible for the scheduler to rank threads in terms of their suitability to run on fast cores.

In order to maintain a common OS design across our asymmetry-aware scheduling policies, we deliberately decoupled the decision of assigning threads of execution to the different core types from the way load balancing is performed[14]. More specifically, while load balancing boils down to deciding *how many threads* must run on fast and slow cores, the specific scheduling policy determines *which threads* in the workload must be mapped to the different core types.

This scheme entails enforcing thread assignments to cores of a given type. Current operating systems provide different mechanisms to perform explicit thread-to-core assignments, which prevent the load balancer from migrating threads onto "non-allowed" cores. Linux's processor affinity masks or Solaris's CPU bindings are examples of such features. Regrettably, explicit mechanisms to bind threads to specific cores tend to cause trouble to the OS load balancer. For example, Linux and Solaris kernels do not maintain separate linked lists of bound and unbound threads, so, in most cases, the load balancer needs to go through the entire run queue to find potential threads to be migrated away from that core, even though many of these threads may not be permitted to execute on the destination core.

In the quest of a lightweight solution to load balancing under these circumstances, we opted to introduce a new abstraction in the operating system: the *core partition*. We define a *core partition* as a set of cores of the same type (either fast or slow). Each core in the system must belong to exactly one partition. Note, however, that there may be more than one partition including cores of an specific type (e.g.,

[14] All scheduling algorithms proposed in this thesis follow the same rules for load balancing but differ in how they perform thread-to-core-type assignments.

several partitions containing slow cores might be present in the system). Given that future many-core systems will contain a very large number of cores, load balancing and accounting may become costly using conventional techniques. In those systems, using core partitions would enable the OS to manage a large number of cores in a scalable way. Henceforth, we will use the term *fast partition* to refer to a partition consisting of fast cores. Conversely, the term *slow partition* will refer to a partition including slow cores.

Load balancing among cores of the same partition can be done according to regular OS policies (e.g., by maintaining per-core run queues evenly loaded). Between partitions, however, load balancing is performed catering to the *scaled load* of individual partitions. The *scaled load* of a core partition, denoted by L_p, is defined as follows:

$$L_p = \lfloor \frac{\sum_{i=1}^{N} run\_queue\_length(C_i)}{\lfloor P \rfloor * N} \rfloor \tag{2.1}$$

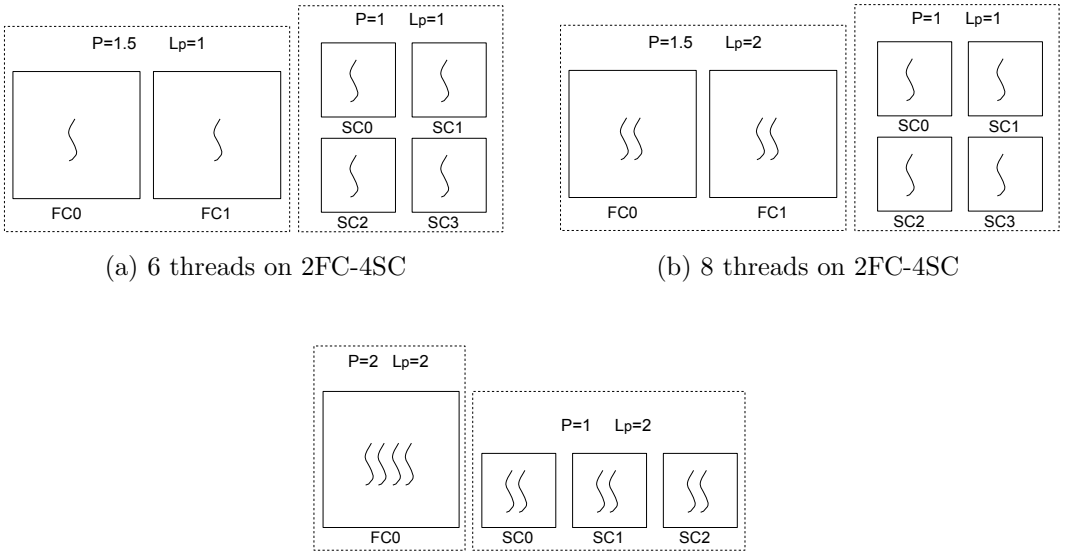
where:

- $C_1, C_2, \cdots, C_N$: cores in the partition.
- N: number of cores in the partition.
- P: scaled computing power of the cores in the partition.
- The function $\lfloor x \rfloor$ denotes the integer part of x. In other words, $\lfloor x \rfloor$ is the largest integer not greater than x.

We define that the system is load balanced if the following conditions hold true:

1. $L_p^{max} - L_p^{min} \leq 1$, where L_p^{max} and L_p^{min} are the maximum and minimum scaled loads among all core partitions in the system, respectively.
2. $L_p^{\mathcal{F}_{min}} \geq L_p^{\mathcal{S}_{max}}$, where $L_p^{\mathcal{F}_{min}}$ is the minimum scaled load among all fast partitions, and $L_p^{\mathcal{S}_{max}}$ is the maximum scaled load among all slow partitions.

Figure 2.4 illustrates these definitions by means of three different examples. In the first scenario (shown in Figure 2.4a), there is one thread per core on an asymmetric system with two fast cores ($P = 1.5$) and four slow cores ($P = 1$) organized into two partitions. The fast partition has two threads ($L_p = 1$) and the slow one has four threads ($L_p = 1$). Thus, the scaled load of each partition is one and the system is load balanced. Figure 2.4b shows the same asymmetric system with two additional threads (8 threads). According to the second rule, the slow partition cannot receive higher loads than the fast partition, so placing four threads on each partition leads



(a) 6 threads on 2FC-4SC

(b) 8 threads on 2FC-4SC

(c) 10 threads on 1FC-3SC

Figure 2.4: Examples of load-balanced asymmetric systems

to a load-balanced distribution. Note that, in the latter case, the fast partition has a higher scaled load than the slow partition. Finally, Figure 2.4c depicts an asymmetric configuration where fast cores have twice as much computation power as slow cores ($P = 2$ for fast cores). In this case, placing four threads in the fast core and two threads on each slow core results in the same scaled load for both partitions ($L_p = 2$). Hence, the system is load balanced.

Intuitively, the scaled load of a core partition, L_p is quite similar to the average scaled load[15] of the cores in the partition. However, we opted to introduce the operator $\lfloor\ \rfloor$ in the formula to avoid situations where partitions with slightly greater computation power than others become unnecessarily overloaded. For example, consider a system with two fast and two slow cores organized into two partitions (fast and slow). Suppose further that the fast cores are 1.5 times faster on average than the slow cores (P is 1.5 for the fast partition and 1 for the slow one) and there are four runnable threads in the system. According to the rules presented above, the load balancer will place one thread per core to ensure load balance. In contrast, if the average scaled load of the cores in the partition were used instead of L_p, the fast partition would receive three threads and the remaining thread would be mapped to a slow core, thus leaving one slow core idle while one fast core remains overloaded.

On the high level, thread-to-core assignments are performed as follows. When a thread enters the system, the scheduler assigns the *idlest* core partition in the sys-

[15]Recall that the scaled load of a core, denoted by L_c, is defined as its run queue length divided by its computation power. Given that cores within a partition $C_1, C_2, \cdots, C_N$ have the same computation power P, the average scaled load of these cores is $\sum_{i=1}^{N} run\_queue\_length(C_i)/(P * N)$.

tem (partition with the lowest L_p) to it. In the event several partitions have the same lowest L_p, fast partitions will be picked first than slow ones to guarantee that Rule 2 holds true. When either load balance rule is not met, thread migrations between partitions are required. While the load balancer performs migrations between partitions of the same type (e.g., between two slow partitions) automatically, migrations between fast and slow partitions are "supervised" by the underlying scheduling policy. We opted to make this design decision because scheduling policies that exploit core specialization techniques are aware of which threads utilize most efficiently the fast cores in the system and, as a result, they are capable of identifying the best *candidate* thread to be migrated out of any partition.

Even when the system is load balanced, migrations between fast and slow partitions may still be necessary. For example, recall that algorithms exploiting TLP specialization may migrate a thread from a slow core to a fast core if it detects that the thread is executing a sequential phase. Under these circumstances, simply enqueueing this thread in a run queue of a core in the target partition could cause load imbalance. To prevent load imbalance, the scheduler *swaps* threads among partitions, rather than performing one-way migrations. This approach entails selecting an appropriate thread to be migrated in the opposite direction. All scheduling policies proposed in this thesis implement different swapping policies, which will be described in detail in Chapters 4, 5 and 6.

2.3.3. Modes of operation

Because symmetric multicore systems will likely coexist with asymmetric ones in the near future, next-generation operating systems will have to deal with platforms consisting of either identical or asymmetric cores. It is also worth bearing in mind that it may be beneficial to temporarily turn an asymmetric system into a symmetric one, since the most efficient mapping for a specific workload could be accomplished by assigning threads on cores of a certain type (either fast or slow) and disabling all unused cores. As a result, asymmetry-aware schedulers must be ready to deal with these hybrid scenarios, where the OS power-management subsystem triggers dynamic transitions between symmetric and asymmetric configurations.

Our scheduling framework has been carefully crafted to be able to cope with these scenarios. To make that possible, the scheduler supports different modes of operation, which rely on core partitions:

- **Asymmetric mode**: This mode is active when cores of different types are enabled at a time in an asymmetric system. One or more fast and slow partitions are present in this mode. Thread-to-core assignments and load-balancing are aided by the underlying asymmetry-aware scheduling policy following the rules presented in Section 2.3.2.
- **Symmetric mode**: This mode gets engaged when all active cores in the system are identical. This is the case of a symmetric-by-design multicore system or when all cores of a specific type on an AMP (either fast or slow

cores) have been turned off. Cores in the system are organized into one or more partitions of the same type. In this case, the underlying asymmetry-aware policy is disabled and thread-to-core assignments and load-balancing are performed using conventional techniques for symmetric systems.

Having different modes of operation makes it possible for the OS to deal with symmetric and performance-asymmetric configurations with a common scheduler design.

2.4. Further challenges

Because conventional operating systems do not account for performance asymmetric hardware resources, scheduling work onto such resources does not ensure an effective utilization of the platform and, at the same time, introduces unpredictability that can limit scalability [17]. Designing strategies to mitigate these issues, delivering the benefits of AMPs to unmodified applications, is a significant challenge for OS developers.

Although information on ILP and TLP gives us an overall idea of which applications will benefit the most from complex, power-hungry cores, further factors contribute to effective and robust thread scheduling on AMPs. Here we will discuss two of them: dealing with functional asymmetry and quality of service (QoS).

As stated in Chapter 1, it is likely that the different core types in future asymmetric multicore processors exhibit certain degree of functional asymmetry (identical in almost every aspect of the ISA but in a small subset of it). In those asymmetric platforms, the system software should assign programs to the different types of cores in consideration with their ISA requirements. In this thesis, we have focused on asymmetric single-ISA multicore hardware, where cores only differ in performance. As a result, the implementation of the proposed scheduling policies may be potentially subjected to runtime failures stemming from the presence of functional differences among cores. Nevertheless, these algorithms can be seamlessly augmented with runtime techniques aimed to deal with those failures, such as the *fault-and-migrate* algorithm proposed in [16].

The specialization techniques described above help scheduling policies to maximize *system-wide* performance. We must emphasize, though, that policies relying solely on these techniques will be inherently unfair in cases where some applications have a higher priority than others or in scenarios where the system needs to deliver QoS guarantees to certain applications. Nevertheless, QoS-oriented policies for AMPs could still leverage knowledge on core specialization in an attempt to provide better service for prioritized applications with a minimal effect on performance. For example, the scheduler might decide to run low-priority CPU-intensive threads on fast cores rather than high-priority highly memory-intensive ones, simply because "wasting" fast cores on running memory-intensive instruction streams may lead to

degrading the overall system performance significantly. In a similar vein, the scheduler may allot a greater share of fast core time to low-priority non-scalable parallel applications and purely sequential ones than to high-priority scalable applications, simply because the former cannot spread their computation across abundant slow cores and the latter will not benefit from running some of their threads on the scarce fast cores. In those scenarios, information on ILP and TLP can be used to make a trade-off between QoS and system-wide performance.

Chapter 3

Experimental Framework

The aim of this chapter is to introduce key aspects of our asymmetric single-ISA test bed, such as the operating system we used to implement our scheduling algorithms and the hardware platforms where we carried out our evaluation. Given that asymmetric multicore systems are not available today, we also provide an overview of the different techniques available for the creation of experimental test beds for doing research on next-generation multicore architectures.

This chapter is organized as follows. Section 3.1 discusses different options available to the industry and to the academy to simulate and emulate asymmetric multicore hardware today. In Section 3.2 we discuss the main reasons that led us to choosing emulation via dynamic voltage and frequency scaling as the foundation for our experimental test bed. Section 3.3 introduces the hardware platforms we used for the evaluation. Section 3.4 presents the OpenSolaris operating system as well as our approach to implementing asymmetry-aware schedulers in its kernel. Finally, Section 3.5 introduces key tools that enabled us to characterize application behavior and helped us deal with the tedious task of system software debugging.

3.1. System software experimentation with AMPs

Although asymmetric single-ISA multicore systems are not being built yet, much of the literature investigating the properties of these systems has originated from major hardware players such as Intel [4, 17, 8] and HP [1, 3, 13]. This fact suggests that within the industry there is interest in this architecture. In anticipation for real-world asymmetric systems, researchers and OS developers can employ different techniques to create experimental test beds for advance work in this area. Nowadays, asymmetric multicore test beds can be based on either *simulation* or *emulation*.

Simulation-based asymmetric platforms devoted to studying OS-level scheduling algorithms for AMPs must support *full-system* simulation. This feature, present in popular software simulators such as M5 [40], Simics [41] or PTLSim [42], enables to

run unmodified OSs on the simulated hardware, which, in turn, makes it possible to measure the overhead incurred by different implementations of the algorithms. Hardware-based simulations are also realizable thanks to publicly available microprocessor specifications, such as the open-source initiative OpenSparc [43]. These platforms potentially enable us to lay out customized multicore designs including cores with different performance and power characteristics into FPGAs. Although these simulation platforms support the execution of an actual operating system as well, most commodity OSs require modest modifications to boot successfully in these systems.

Emulation techniques, on the other hand, enable to turn off-the-shelf symmetric multicore hardware into asymmetric. This can be accomplished by various means. For example, dynamic voltage and frequency scaling (DVFS) techniques make it possible to have cores with different performance and power consumption on existing chip multiprocessors. To this end, some cores in the system can be configured as "fast" by making them operate at the highest DVFS level, whereas the remaining cores are slowed down – giving rise to "slow cores" which operate at a lower frequency and consume less power. Asymmetric configurations with similar differences in performance between cores to those DVFS-based, can also be obtained by varying the duty cycle of the processor [4].

The emulation of asymmetric systems with greater microarchitectural differences between cores is also possible. This can be accomplished by changing certain microarchitectural characteristics in some of the cores, ranging from the pipeline issue/retirement width to the cache size or the number of active functional units. With this approach, symmetric multicore systems are made asymmetric by diminishing certain capabilities of some cores. Finally, asymmetric hardware can also be built as a multiprocessor system with disparate processors in each socket [8].

3.2. Emulation of asymmetric single-ISA multicore hardware via DVFS

The scheduling policies proposed in this thesis take advantage of different core specialization techniques targeting a wide range of workloads. In order to carry out a thorough evaluation of these policies in diverse scenarios and assess the overhead of the different implementations in the OS, the chosen testbed must meet two requirements. First, it is desirable that the testbed in question provides affordable simulation times when experimenting with many cores. This is necessary to test complex workload mixes –potentially including parallel applications–, as well as to make it possible to detect potential scalability bottlenecks that the scheduler may exhibit as the number of cores increase, thus enabling us to devise potential remedies. Second, the testbed must also allow us to perform experiments with commodity operating systems and rely solely on tools available to academic research.

Emulating AMPs by introducing performance asymmetry in existing symmetric multicore hardware was the most straightforward approach to meet our needs. In

this thesis, more specifically, we opted to introduce performance asymmetry by slowing down the frequency of some of the cores in the system via Dynamic Voltage and Frequency Scaling (DVFS). This technique combines rapid emulation of medium- to large-scale asymmetric single-ISA systems, with the capability of running an actual OS with native performance. Not only does DVFS enable to vary the performance of the cores, but it also helps to substantially reduce their energy/power consumption by adjusting both their voltage and frequency levels [27, 44, 45]. Therefore, by means of this technique we can effectively emulate AMPs including cores with different performance and power characteristics[1].

Most state-of-the-art chip multiprocessors feature DVFS management capabilities, ranging from low-power processor designs such as the Intel XScale or the AMD Mobile K6 Plus processors, to high-performance server processors such as the AMD Quad-Core Opteron or the Intel Quad-Core Clovertown and Core i7 processors [46]. In some DVFS implementations, such as in the latest Intel processors, several cores on the same chip or physical package share the same power domain, so they must operate at the same DVFS level (same frequency). Conversely, other processors implement *core-level* DVFS via multiple frequency domains, which make it possible to change the frequency of each core independently. The first general-purpose multicore processor to support a form of core-level DVFS[2] was the AMD Quad-Core Opteron [48]. On this system, cores with different frequency may still have the same voltage.

The other simulation/emulation techniques mentioned in the previous section were ruled out since they did not simply meet the requirements demanded by our experimental evaluation. First, current full-system simulation platforms available to the academia for research on multicore processors –such as PTLSim or Simics– do not deliver affordable simulation times when testing complex workload mixes including parallel applications, which requires simulating a rather large number of cores[3]. In a similar vein, FPGA-based simulation platforms enable to synthesize only a reduced number of cores on the type of reconfigurable hardware accessible to the academia at the present time. This shortcoming prevents this simulation technique from being suitable for experiments with parallel applications. Therefore, with current full-system simulation technology is unrealistic to carry out simulations of medium/large scale asymmetric multicore systems, such as the ones we need for the evaluation of the scheduling algorithms proposed in this thesis.

On the other hand, techniques that introduce performance asymmetry by diminish-

[1]When downscaling the frequency of a core by means of DVFS, its performance may decrease linearly with the frequency at the most (see Figure 2.1), while a cubic benefit in power consumption is achieved due to voltage scaling.

[2]Currently, multiple on-chip voltages are provided by off-chip voltage regulators, which are bulky and costly, and usually require several microseconds to perform a change in the voltage. Recent work in this area attempt to address these limitations by proposing several designs of on-chip regulators, which allow to perform voltage changes in nanoseconds rather than in microseconds, and consume less power than off-chip implementations [47].

[3]For instance, the simulation speed of PTLSim for a homogeneous 8-core SMP with detailed out-of-order simulation is around 100-200 processor cycles per second on a 2.4 GHz server processor (for simple-application workloads from Splash-2).

Table 3.1: Main features of the Intel-8 target platform.

Server Model	Dell PowerEdge 2950	
Processor	2 x Quad-Core Intel® Xeon® X5365 CPU @ 3.0GHz	
	Total cores	8
	Topology	2 chips, 2 physical packages per chip 2 cores per physical package sharing an L2 cache
	L1 cache	32KB+32KB (data+instruction), private
	L2 cache	4MB shared (unified), shared
Memory	8 GBytes (4x2GB) DDR2-667MHz SDRAM UMA architecture	
DVFS capabilities	4 DVFS Levels 2000MHz, 2333MHz, 2667MHz and 3000MHz Cores in the same physical package share DVFS domain	

ing the microarchitectural capabilities (other than the processor frequency) of some cores in current multicore hardware would make it possible to emulate AMPs as efficiently as via DVFS. Unfortunately, they are only fully realizable in the industry. This stems from the fact that changing microarchitectural characteristics on commercial processors, such as the pipeline retirement width or the cache size, can be only effected via proprietary tools, which are not available to the academia [8]. In recent papers from Intel, some details about actual prototypes created by means of proprietary tools have been disclosed [24]. Regrettably, these prototypes are not available for academia research at the present time.

Building asymmetric hardware as multiprocessor systems with disparate processors in each socket raises significant barriers as well. First, creating systems of this type requires to develop a customized BIOS to get the system to boot successfully. This restriction makes the approach inaccessible to the academic community as well, since BIOSs' source code is only accessible within the industry. Second, this approach does not allow to create asymmetric systems with fast and slow cores in the same chip (i.e., sharing a last-level-cache), which, as we will see later (see Section 5.2.4), would enable to perform inexpensive migrations between different core types and, as a result, effectively mitigate migration overheads.

3.3. Experimental setup

Our evaluation was carried out on three multicore server systems: (1) *Intel-8* – an 8-core machine with two Intel "Clovertown" Quad-Core chips; (2) *AMD-8* – an 8-core system including two AMD Quad-Core "Barcelona" chips; and (3) *AMD-16* – a 16-core system consisting of four AMD Quad-Core "Barcelona" chips. More details about the processor, memory architecture and DVFS capabilities of the *Intel-8*, *AMD-8* and *AMD-16* systems can be found in Tables 3.1, 3.2, and 3.3, respectively.

Our asymmetric configurations consist of two core types: "fast" and "slow". In

Table 3.2: Main features of the AMD-8 target platform.

Server Model	Sun Microsystems Sun Fire X2200 M2	
Processor	2 x Quad-Core AMD® Opteron® 2354 CPU @ 2.2GHz	
	Total cores	8
	Topology	2 chips 4 cores per chip sharing an L3 cache
	L1 cache	64KB+64KB (data+instruction), private
	L2 cache	512KB (unified), private
	L3 cache	2MB victim cache (unified), shared
Memory	16GB in two banks of 8 GBytes (4x2GB) each DDR2-533MHz SDRAM NUMA architecture (Hypertransport-based)	
DVFS capabilities	5 DVFS Levels 1100MHz, 1400MHz, 1700MHz, 2000MHz and 2200MHz Per-core adjustable DVFS levels	

Table 3.3: Main features of the AMD-16 target platform.

Server Model	Dell PowerEdge 2950	
Processor	4 x Quad-Core AMD® Opteron® 8356 CPU @ 2.3GHz	
	Total cores	16
	Topology	2 chips 4 cores per chip sharing an L3 cache
	L1 cache	64KB+64KB (data+instruction), private
	L2 cache	512KB (unified), private
	L3 cache	2MB victim cache (unified), shared
Memory	64GB in four banks of 16 GBytes (8x2GB) each DDR2-533MHz SDRAM NUMA architecture (Hypertransport-based)	
DVFS capabilities	5 DVFS Levels 1150MHz, 1450MHz, 1750MHz, 2050MHz and 2300MHz Per-core adjustable DVFS levels	

this thesis, we use the "xFC-ySC" notation to refer to different asymmetric configurations, where x and y denote the number of fast and slow cores in the platform, respectively. The chosen frequencies for fast and slow cores remain the same across the asymmetric configurations associated to a given machine. Since we wanted to get the most asymmetric setting out of our platforms, the frequency of fast and slow cores was set to the maximum and minimum frequency (DVFS) levels, respectively. In particular, on Intel-8's asymmetric configurations, fast cores operate at 3.0 GHz, while slow cores run at 2.0 GHz (fast cores are 1.5 times faster than slow cores). On the AMD platforms, in contrast, higher performance differences between core types are obtained since fast cores operate at twice the frequency of slow cores.

The AMD-8 and AMD-16 platforms, which are based on the Opteron "Barcelona" processor, support core-level DVFS, so we are able to vary the frequency for each core independently. Thanks to this flexibility, we can create many different asymmetric configurations, even with fast and slow cores sharing a last-level cache. The

Intel-8 platform, on the other hand, does not exhibit this flexibility[4].

The memory architecture of the platform has important implications on the performance of our schedulers. In this thesis, we explore two different memory architectures: an FSB-based UMA system (Intel-8) and two NUMA platforms (AMD-8 and AMD-16). Previous researchers observed (and so did we) that the overhead associated with a thread's migration can be significant for certain applications in UMA systems, but it tends to be more significant in NUMA platforms [17]. This stems from the fact that access to a local memory bank on the latter kind of systems incurs a shorter latency than access to a remote memory bank. We will elaborate on these issues in Section 5.2.4.

Finally, it is worth noting that the processor frequency may change dynamically due to the action of power-management mechanisms triggered by the OS. As explained in Section 2.3.1, the OS may decide to lower the DVFS level of certain cores to save power (e.g., those that have been idle for a while), which results in a lower processor frequency and performance. Because our experimental asymmetric platform is based on performance asymmetry due to processor frequency, arbitrary changes in the frequency of the cores would lead us to drawing misleading conclusions from our experiments. Furthermore, these changes violate one of the main assumptions of our experimental platform: the ratio between the maximum frequencies of fast and slow cores does not change dynamically. To prevent that from happening, we opted to disable some power-management features in the Solaris kernel[5]. Nevertheless, we strongly believe that power-management will remain as a key factor towards efficient utilization of future multicore hardware, so the study of performance asymmetry coupled with dynamic changes in the features of the different cores is an interesting avenue for future work.

3.4. Operating system: OpenSolaris

The implementation of our scheduling algorithms was carried out in the Solaris kernel. More precisely, we used the b86 and b111b versions of the kernel, included in the last two stable versions of the OpenSolaris operating system: OpenSolaris 2008.05 and OpenSolaris 2009.06, respectively.

Although the vast majority of the results reported in this thesis correspond to the implementation in b86, the algorithms have been recently ported to b111b to be evaluated in new architectures not fully supported by b86, such as the Intel Nehalem and Intel Westmere processors. The b111b version includes significant changes with respect to b86, but these changes did not affect much the scheduling subsystem and,

[4] On the Intel Clovertown processor, cores in the same physical package, which share a last-level cache, are also within the same power domain. As a result, they must operate at the same voltage/frequency level.

[5] For example, the Solaris's Power-Aware Dispatcher [49] may set idle cores into a deep low-power state, where no instructions can be executed. Usually, the process of transitioning this core into a state that allows the execution of instructions takes some time, and as a result performance variations can be observed.

as a result, the implementations of our schedulers in both OS versions differ very little from each other. Furthermore, we observed that the performance delivered by both versions was quite similar on our Intel-8 and AMD-8/16 platforms.

The remainder of this section, divided into four subsections, is devoted to presenting some key aspects of OpenSolaris. In Section 3.4.1, we briefly describe the history of OpenSolaris. In Section 3.4.2, we outline the structure and main components of the scheduling subsystem in the Solaris kernel. The key factors that led us to choosing OpenSolaris to host our experimental framework are highlighted in Section 3.4.3. Finally, in Section 3.4.4, we present our approach to implementing asymmetry-aware scheduling algorithms taking advantage of the existing infrastructure in the Solaris kernel.

3.4.1. Brief history of OpenSolaris

OpenSolaris was a descendant of the UNIX System V Release 4 (SVR4) codebase developed by Sun and AT&T in the late 1980s. Currently, it is the only version of System V available as open source. OpenSolaris was created by Sun Microsystems in 2007, out of a combination of several software consolidations which were open sourced shorty after the release of Solaris 10.

The OpenSolaris project was created by Sun to build a developer and user community around Solaris, but more importantly, it was meant to be a test bed for Solaris, Sun's commercial operating system. In other words, Sun planned that future versions of Solaris would be based on the OpenSolaris project. Regrettably, soon after the acquisition of Sun Microsystems in 2010, Oracle decided to stop releasing and discontinue the OpenSolaris distribution and the development model.

Following Oracle's discontinuation, a group of former OpenSolaris developers decided to take over by creating the Illumos project, a fork of the ON (Operating system and Networking) package, which included the kernel and a key set of user-space tools of OpenSolaris. In turn, the OpenIndiana project, a part of the Illumos Foundation, aims to continue OpenSolaris development and distribution, using Illumos as its core. On September 24, 2010, the first release of OpenIndiana came out, giving rise to the first open source distribution based on OpenSolaris after the end of the project.

3.4.2. Scheduling subsystem in OpenSolaris

The core subsystem that takes care of thread scheduling in the Solaris kernel has a modular design. On the high level, this subsystem consists of two components: the *dispatcher* and the *scheduling modules*.

The kernel dispatcher is the code that places runnable threads on a dispatch queue (run queue), selects the next thread to run on a processor, and manages the switching of threads on and off processors. The Solaris kernel implements a *global priority*

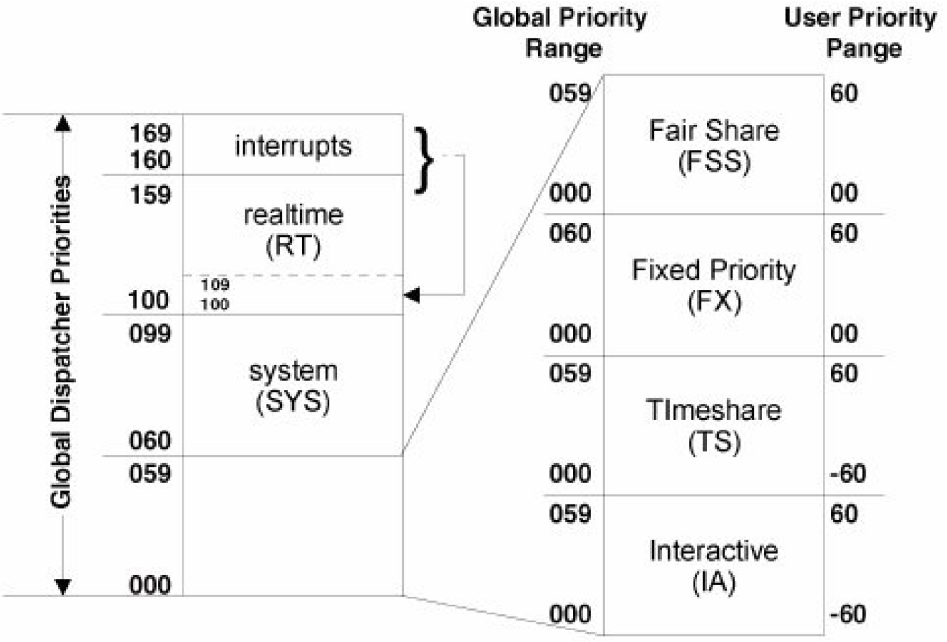


Figure 3.1: Dispatcher global priorities.

scheme, which guarantees that a runnable thread's global priority solely determines how soon it will be selected to run. The dispatcher makes this possible by enforcing that threads with the highest priorities run first than other threads at any time. Runnable threads are assigned a global priority ranging from 0 to 169.

Every thread in the system is in one of several possible scheduling classes; this arrangement determines the range of priorities for the thread, as well as which class-specific scheduling algorithms will be applied as the thread runs. As shown in Figure 3.1, the global priority range is divided into several priority subranges associated to different scheduling classes. Multiple scheduling classes provide a powerful and flexible mechanism for managing various workloads with different scheduling requirements and making efficient use of the system's processors. The current version of the Solaris kernel supports five scheduling classes available to user applications: Timeshare (TS) – default class –, Interactive (IA), Fair Share (FSS), Fixed Priority (FX) and Real Time (RT). The SYS class, also shown in Figure 3.1, is reserved by the kernel for the execution of operating system threads.

In the Solaris kernel, scheduling classes are implemented as separate, dynamically loadable scheduling modules. While the dispatcher is in charge of key tasks such as load balancing and global runqueue management, the scheduling modules handle CPU accounting, time-slice expiration and priority boosting. The scheduling class of a thread is inherited from the parent process, but it can be changed afterwards via the `priocntl` administrative command or by invoking the system call with the same name.

3.4.3. Why OpenSolaris?

Over the course of history, complexity faced those attempting to understand why a system was not meeting its prescribed service-level and response-time goals. The problem was that the performance analyst had to work with only a small set of hardwired performance statistics, which, ironically, were chosen some decades ago by kernel developers as a means to debug the kernel's implementation. As a result, performance measurement and diagnosis became an art of inferencing and, in some cases, guessing. Whether the underlying problem has to do with process misbehaving, performance degradation, system panics, or hardware failure, there is one key element to any of these: *if you cannot observe the problem, you cannot fix it.*

Today, OpenSolaris has a rich set of observability facilities, aimed at the administrator, application developer, and operating systems developer. The set of commands, tools and utilities that make up the observability framework can be categorized in terms of the information they provide and the source of the data. They include the following:

- **Kernel-statistics-gathering tools**: Report kernel statistics (*kstats*), collected by means of counters. Examples of commands within this category are `vmstat`, `mpstat`, and `netstat`.
- **Process tools:** Provide system process listings and statistics for individual processes and threads. Examples are `prstat`, `ptree`, and `pfiles`.
- **Forensic tools:** Track system calls and perform in-depth analysis of targets such as applications, kernels, and core files. Examples are `truss`, and MDB.
- **Dynamic tools:**. Fully instrument-running applications and kernels. DTrace is an example.

Among all the observability tools, MDB and Dtrace deserve special attention when it comes to kernel debugging. Both tools provide us with various means to observe any part of system and application behavior, ranging from every instruction in an application to the depths of the kernel. The fact that these powerful tools were available in OpenSolaris was the main reason that led us to choosing this operating system for the study of scheduling algorithms for asymmetric multicore systems. A brief introduction to Dtrace and MDB can be found in Sections 3.5.2 and 3.5.3, respectively.

Apart from OpenSolaris's comprehensive observability framework, key features of the kernel make this OS suitable to accommodate our scheduling algorithms. The first stunning property of the design of Solaris kernel is *scheduler isolation*, which makes it possible to have multiple scheduling classes (algorithms) in the system. Each scheduling class makes scheduling decisions for a specific set of threads (threads associated to this class) as if they were alone in the system. As an illustrative example, consider that there are four runnable threads in the system, two of them assigned to the FX class and the other two are system threads (SYS class). In this

scenario, the FX scheduling policy "sees" just two runnable threads and, as a result, its scheduling decisions target these two threads only.

We found that scheduler isolation is of remarkable importance when doing research on OS scheduling using real hardware, since it enables us to choose the threads that will be actually scheduled by our algorithm. Likewise, this kind of isolation mechanism prevents our scheduler from reacting to the presence of "non-desired" threads, which happen to become runnable during our experiments. Note that, even when the machine is fully dedicated to experiments, periodic short-lived threads such as system daemons or kernel threads may become runnable every so often (e.g., reacting to a timer interrupt). In the Solaris kernel, these special threads are handled by different scheduling classes from the one that implements our new scheduling algorithm. All in all, we observed that this feature leads to greatly simplifying the scheduler implementation.

The ease of incorporating new scheduling algorithms into the system is another appealing property of the Solaris kernel. As we mentioned above, scheduling algorithms are implemented as separate scheduling classes, which are deployed as independent, dynamically loadable modules. With this framework, the process of incorporating a new scheduling class into the system boils down to developing the associated scheduling module. Because getting to work a new scheduling class does not require any changes to the dispatcher, deploying a new scheduling class in OpenSolaris turns out to be a fairly straightforward task. Notably, this is not the case of other open-source kernels that also support multiple scheduling classes, such as Linux. Despite the recent efforts aimed to make the Linux kernel's scheduling subsystem modular (starting from the 2.6.23 version), incorporating a new scheduling class into the Linux kernel today entails making significant changes to the core part of the scheduler[6].

3.4.4. Implementing asymmetry-aware policies in the Solaris kernel

Solaris's scheduling modules are a powerful mechanism to seamlessly augment the operating system with new scheduling algorithms. Unfortunately, the current interface between the dispatcher and the scheduler classes restricts the potential actions that a scheduling module can perform. For example, certain operations like selecting a specific CPU for a thread to run on or controlling the load of a set of CPUs are restricted to the dispatcher code, which has access to per-CPU run queues. This limitation makes it impossible for a conventional scheduling module alone to implement asymmetry-aware scheduling algorithms, since key actions such as forcing that a thread runs on a particular core type (fast or slow), or allowing that fast cores receive higher loads than slow cores cannot simply be performed from the module's code.

[6]This stems from the fact that a fairly significant amount of source code in the Linux scheduling subsystem (mostly included in the `sched.c` file) makes explicit reference to the scheduling classes available: Real Time (RT) and fair (CFS).

To overcome these shortcomings, we introduced a set of callbacks in the dispatcher code which allow asymmetry-aware scheduling modules to work cooperatively with the dispatcher when making load balancing decisions and thread-to-core assignments. On the high level, our scheduling modules are notified whenever the dispatcher selects a CPU for a thread to run on, as well as when a thread must be migrated from a core to another to enforce load balance (a.k.a. *thread stealing* operations). Upon notification, the module can select a target CPU different from the dispatcher's choice, thus avoiding "non-desired" thread migrations. Notably, the modifications involved in introducing the aforementioned callbacks into the dispatcher's code were astonishingly modest: as little as twenty lines of new source code were necessary.

Yet, enhancing the interface between the scheduling class and the dispatcher alone is not sufficient to guarantee an efficient implementation of asymmetry-aware policies exploiting core specialization techniques (just like the ones we explore in this thesis). The fact that these policies tend to make a unequal allotment of fast-core cycles to threads (e.g., catering to the relative benefit threads derive from running on a fast core over a slow one), gives rise to significant challenges related to load balancing.

In the quest of efficient load balancing under these circumstances, we devised a scheme based on *core partitions* (presented in Section 2.3.2). In a nutshell, all the cores in the system are organized into *fast* and *slow partitions* – non-overlapping sets of cores consisting of fast and slow cores, respectively. The asymmetry-aware scheduler performs thread assignments to core partitions (sets of cores) rather than to individual cores. The partition-based approach makes it possible for the operating system to perform load balancing among cores of the same type (i.e., *intra-partition* load balancing) by means of conventional techniques used on symmetric systems; and at the same time, lets the asymmetry-aware scheduler decide how to distribute the load across different core types (i.e., *inter-partition* load balancing).

Among the resource management controls in the Solaris kernel, *processor sets* enable us to implement core partitions very efficiently. Before explaining how this is done, we introduce Solaris's processor sets.

Processor sets allow cores on a multiprocessor/multicore system to be partitioned into groups or sets, where each set has one or more cores assigned to it. When the system boots, the kernel creates a *default* processor set including all the cores in the system. Additional processor sets can be created using administrative tools[7]. Having different processor sets allows the system administrator to safely consolidate several CPU-bound workloads onto a single server by reserving groups of cores or processor sets, for the execution of specific applications.

In the Solaris kernel, all threads have a processor set assigned to it, which is inherited from the parent process but can be changed afterwards via administrative tools. The dispatcher enforces that runnable threads do not get scheduled on cores

[7]Processor sets are dynamic: the creation and deletion of sets, adding and removing processors to and from sets, and process/thread binding are all done without requiring a system reboot. As many processors sets as needed can be created, as long as one processor remains in the default processor set to run operating system kernel threads.

not belonging to the thread's processor set. Load balancing in the Solaris kernel is specifically tailored to work efficiently with processor sets. To this end, the dispatcher performs load balancing independently on each processor set by attempting to keep the run queues of cores within a processor set as evenly loaded a possible. Hence, load balance is guaranteed among cores in the same processor set, but not across processor sets.

In our implementation, we performed a one-to-one mapping between processor sets and partitions. Since future many-core systems may contain a very large number of cores, load balancing and CPU accounting may become costly. In this scenario, having fast and slow cores organized into more than one partition per core type may turn out beneficial, since multiple partitions would enable to manage a large number of cores in a scalable way. The evaluation of our scheduling policies, however, was performed using two core partitions only: one fast and one slow partition, containing all fast and all slow cores in the system, respectively. We opted to do so because the maximum core number of our emulated AMPs (sixteen cores) is still far from the number of cores expected for next-generation many-core systems [50]. Nevertheless, studying how the scalability of different implementations vary with the maximum size of a partition (number of cores in it) is an interesting avenue for future work.

Our partition-based asymmetry-aware scheduling modules perform thread mappings to specific core types (fast or slow) quite simply by assigning the appropriate processor set to each thread (associated to the fast or the slow partition, respectively). Given that partitions are implemented as processor sets, intra-partition load balancing is carried out automatically by the dispatcher. Inter-partition load balancing, however, is totally up to the asymmetry-aware scheduling module. As explained earlier in Section 2.3.2, the approach to inter-partition load balancing followed by all the scheduling algorithms proposed in this thesis, relies on a set of rules indicating when the system is load balanced. In the event any of these rules is not met, the asymmetry-aware scheduling module is responsible for triggering as many thread migrations between partitions as necessary to enforce load balance.

Whether core partitions must be created manually or populated automatically by the kernel when it boots, depends on whether the asymmetric system is asymmetric by design or emulated. In future asymmetric multicore systems, the operating system must detect the different core types during the boot process and populate core partitions accordingly at that point. In our emulation-based experimental platform, however, the asymmetric configuration –caracterized by its number of fast and slow cores, fast-to-slow frequency ratio and topology– can be created after the boot process, and we can even switch to a different configuration later on without rebooting the machine[8]. This dynamic capability allows us to perform several experiments in a row with virtually any asymmetric configuration the machine supports. Nevertheless, to take full advantage of this feature, the scheduling module must be aware of these dynamic changes to properly update its internal structures. To this

[8]Transitions between different voltage/frequency levels do not require a system reboot to take effect. Likewise, most off-the-shelf operating systems include command-line administrative tools enabling the system administrator to change the number of active cores on the system, which do not require a system reboot either.

end, our experimental framework supports dynamic reconfiguration of fast and slow partitions when changes in the asymmetric configuration take place.

3.5. Other tools

3.5.1. MICA

A well-known way of tracing the behavior of a program is through binary instrumentation, which allows arbitrary code to be executed during predefined points in the program as it runs. In this thesis, we used Pin, a binary instrumentation framework from Intel [51]. Pin is non-invasive, in the sense that it does not require any modifications of the target program for instrumentation purposes. In addition to what is supported by other state-of-the-art instrumentation tools, Pin allows custom toolkits to be built on top of it, also known as *pintools*.

One of such pintools we used very extensively is MICA (Microarchitecture-Independent Characterization of Applications), created by Hoste and Eeckhout [52]. MICA allows the user to collect a number of program characteristics to quantify runtime program behavior. The statistics gathered are *microarchitecture independent*, which are not affected by details of the underlying hardware. Metrics such as opcode (instruction) mix, memory access patterns, register access patterns or the amount of inherent ILP, are examples of microarchitecture-independent metrics. These metrics stand in contrast with *microarchitecture-dependent* metrics, such the cache miss rate, which may vary substantially across different cache configurations with different hierarchy levels and sizes, or the number of instructions per cycle (IPC), which is intimately connected with the clock speed and the pipeline implementation.

Previous research has demonstrated that microarchitecture-dependent metrics do not give an abstract picture of application performance since they may greatly vary across systems [53, 54, 55]. As a result, workload characterization techniques based on program characteristics totally independent of the microarchitecture (cache configuration, branch predictor, ...) on which the measurements are done, usually provide better a classification of applications than other techniques relying on simulation or hardware performance counters. In particular, the extraction of certain microarchitecture-independent metrics with MICA out of an application's execution on one machine, enables us to accurately predict the cache behavior of the application across different platforms differing in cache hierarchy and size (as we will explain in Section 4.1.1). Therefore, thanks to MICA, selecting a comprehensive set of workloads with diverse behavior across systems turns out to be a much more simple task.

3.5.2. Dtrace

Current operating systems include a comprehensive set of tools to monitor various aspects of the system, ranging from the amount of memory used by user applica-

tions, to the share of the total network bandwidth each application is demanding. As rich as each individual tool is, it still provides only limited and fixed insight into one specific area of a system. More importantly, most tools provided by the system display data in different formats and frequently have very different interfaces, which results in a bunch of tools in disjoint operation. Therefore, accurately correlating the events reported by a set of monitoring tools with the specific parts of the application(s) that are driving the behavior the tools report becomes a challenging task.

Dtrace, OpenSolaris's dynamic tracing infrastructure, makes this issue a thing of the past. This powerful tool enables to observe any part of system and application behavior, ranging from every function call in an application to the depths of the kernel. The fact that Dtrace provides a single interface to this vast array of information makes it possible to easily observe cause and effect across the entire software stack, allowing that subsystem boundaries can be crossed seamlessly. For example, requests such as "identify the applications that caused writes to a given device" or "display the kernel code path that was executed as a result of a given application function call" are now trivial to fulfill.

Before carrying on with a brief introduction to Dtrace, it is worth highlighting that the aim of this section is not to go into great detail on the language and architecture of the dynamic tracing infrastructure, but to outline the basic operation of Dtrace and to show its versatility. A thorough treatment of Dtrace can be found in [56] and [57].

DTrace has its own scripting language which enables us to express the questions we want to ask; this language is called "D". It provides most of the richness of "C" plus some tracing-specific additions. D is a block-structured language similar in layout to `awk`. A D program consists of one or more clauses that take the following form:

```
probe
/ optional predicates /
{
        optional action statements;
}
```

Each clause describes one or more probes to enable, an optional predicate, and any actions to associate with the probe specification. For example:

```
syscall::read:entry
{
        printf("I'm inside read()");
        exit(1);
}
```

The above script contains one clause which enables the read(2) system call entry probe. When this script is executed, the system is modified dynamically to insert

our tracing actions into the read() system call. When any application next makes a read() call, the clause is executed, causing the string "I'm inside read()" to be displayed. The exit(1) call terminates the tracing session, an action that in turn causes the enabled probes and their actions to be removed. The system then returns to its default state. After executing the script consisting of the latter code (`example.d`), we will see this:

```
opensol# dtrace -q -s example.d
I'm inside read()
```

To accomplish its goals, Dtrace relies on dynamic modification of application and kernel code. On the high level, this mechanism works as follows. The D script we specify is transformed on the fly into a machine-independent code called DIF (D Intermediate Format), which is stored in memory regions inside the kernel's address space. Based on the probe descriptions, the dynamic framework detects those parts in the application and kernel code where the actions specified in the D script may need to be invoked; the evaluation of the predicate associated to each probe will actually determine at runtime if this is necessary.

In the event the probe fires (i.e., the associated predicate holds true), the kernel executes the DIF code by means of a DIF-capable virtual machine, in much the same way as a Java virtual machine interprets Java bytecode. Using a virtual machine for this task makes it possible to execute arbitrary code safely on production systems without inducing failure. The use of a runtime emulation environment ensures that errors such as dereferencing null pointers can be caught and dealt with safety. In this aspect, Dtrace stands in contrast with other popular system's tracing frameworks, such as Linux Kprobes [58], which also enable to specify similar probes but by means of custom loadable kernel modules. Because these custom loadable modules are written in raw C code, it is quite possible to crash the machine if the instrumentation code has a flaw in it. As a result, this kind of tracing framework is not suitable for production systems.

The modifications made to the system in response to the execution of a D script (the "instrumentation") exist just for the lifetime of the script. To make that possible, Dtrace triggers a set of actions at the end of the tracing session causing the enabled probes and the generated DIF code to be removed from the kernel. Therefore, when no D scripts are running, the system acts just as if DTrace were not installed; thus avoiding overheads that otherwise may degrade the system performance.

Throughout the evaluation and development of the scheduling algorithms proposed in this thesis, Dtrace enabled us to shed light on key aspects of application-scheduler interaction. For example, keeping track of the amount of execution time the asymmetry-aware scheduler mapped each application in a workload to fast and slow cores was easy to accomplish by means of a simple D script. Dtrace also allowed us to characterize parallel applications in terms of their amount of sequential portion, which was essential to select a representative, wide range of workloads to evaluate the PA and the CAMP scheduling algorithms. To this end, we wrote a D

script to measure the amount of time the application spent with a single runnable thread. (Note that this information is visible from the OS code only.) Because of Dtrace's capability to seamlessly cross boundaries between different levels of the software stack, we could also tell whether runnable threads where actually doing useful work or busy waiting (*spinning*) at a user-level synchronization primitive in the threading library or runtime system.

3.5.3. MDB

Software program failures can be broadly divided into two categories: problems that can be solved with source-level debugging tools; and problems that require low-level debugging facilities, examination of core files, and knowledge of assembly language to diagnose and correct. The former category of problems can be solved by means well-known features, ranging from step-by-step execution or explicit breakpoints, to the capability to display program variables as the program runs; these features are widely supported by popular debuggers, such as `gdb`. However, when programming a complex low-level software system such as an operating system, problems of the latter class frequently arise. The Modular Display Debugger (MDB) facilitates analysis of these "not-so-conventional" situations [56].

MDB provides a powerful set of built-in commands enabling to analyze the state of programs at the assembly language level. It also includes a dynamic module facility that programmers can use to implement their own debugging commands to perform program-specific analysis.

The OpenSolaris operating system includes a set of MDB modules that assist kernel programmers in debugging the Solaris kernel and related device drivers. These MDB modules facilitate sophisticated analysis of kernel and process state, in addition to standard data display and formatting capabilities. The debugger modules allow you to formulate complex queries to do the following:

- Locate all the memory allocated by a particular thread.
- Determine what type of structure a particular memory address refers to.
- Locate leaked memory blocks in the kernel.
- Analyze memory to locate stack traces.

With MDB, operating system developers can retrieve any information about internal structures of a running kernel, such as thread descriptors, run queues or the hierarchical topology of the system. MDB also enables us to retrieve this information from kernel core files, saved from a prior system crash. Notably, one of MDB's most striking features is the capability to be executed interactively right after a system crash or even in severe error conditions that do not lead directly to a system crash (e.g., when the system is deadlocked), thus allowing to look into the problem right when it happens.

Chapter 4

Catering to the Microarchitectural Diversity of the Workload

Previous research has demonstrated that different core specialization techniques contribute to improve efficiency of asymmetric multicore systems in diverse scenarios [1, 3, 2, 13]. This chapter focuses on *ILP specialization*, which caters to the microarchitectural diversity present in the workload, namely, the diversity in the relative benefit that the threads in the workload derive from running on a fast core rather than a slow one. As pointed out in Section 2.1.1, the exploitation of this specialization technique enables asymmetric single-ISA multicore systems to deliver higher performance per watt than symmetric ones. For example, threads running memory-intensive code should typically be mapped to slow cores, because the speedup they experience on fast cores relative to slow cores is disproportionately smaller than the additional power that the fast cores consume.

Efficiency of asymmetric systems is maximized when applications are matched to cores according to the architectural properties of both. This matching can be conveniently performed by an operating system thread scheduler. In this chapter we describe a new asymmetry-aware scheduling algorithm that employs an original methodology compared to the ones proposed in the past. Our algorithm, called Heterogeneity-Aware Signature-Supported (HASS) scheduler is based on the idea of *architectural signatures*. An architectural signature is a compact summary of architectural properties of an application. It may contain information about the application's memory access patterns, instruction-level parallelism (ILP), sensitivity to variations in the clock speed and other data. The key property of this information is that it can be efficiently interpreted by the scheduler to determine how well a given application "matches" a given core.

The architectural signature framework evaluated in this thesis is designed for asymmetric systems where cores differ in the clock speed since such a system can be emulated very efficiently using existing multicore processors (as shown in Section 3.2). To capture the properties of the application that determine its sensitivity to variations in these architectural features, the architectural signatures are based on an application's memory intensity. Memory intensity is captured by the thread's

cache miss rate, which as we found can be used to model the performance of the application on cores with different clock speeds. We implemented two versions of HASS, static (HASS-S) and dynamic (HASS-D). With the static version the architectural signature is constructed offline. In this case we obtain the application's *reuse-distance profile* [59] (a summary of the memory reuse patterns) and use it to estimate the miss rate for caches of various sizes and associativities to cater to possible systems where the application may run. With the dynamic version, the miss rate is measured online, using hardware performance counters. Performance estimates generated using cache miss rates can be used to compute the relative benefit that an application (or thread) derives from running on different cores. By comparing the relative benefits for different threads, the scheduler decides which thread is the best candidate for a particular core type.

Our architectural signature framework can be generalized to systems where cores differ in other microarchitectural features, but exploring such architectures was outside the scope of this thesis. Instead, we chose to address the systems that could be effectively emulated on existing hardware (as opposed to on simulators), because this enabled us to perform a more extensive and thorough evaluation than what would have been possible on a simulator[1].

Architectural signatures allow *estimating* the relative performance of threads on cores of different types. Another alternative is to measure this performance directly, by running each thread on each possible core type. One goal of our work was to compare HASS to an algorithm based on that approach, and we were aware of two previously proposed algorithms that relied on it. They determined the best matching of threads to cores via an online performance monitoring mechanism that required running each thread on each core type [2, 3]. When we implemented one of these algorithms (IPC-Driven [2] proposed by Becchi et al.), we found that this performance monitoring methodology often leads to an incorrect estimate of the relative benefit that a thread derives from running on a particular core due to the dynamic nature of program phases. In addition, the necessity to periodically re-measure threads' performance on different cores creates imbalanced demand for cores of different types if there are more cores of one type than of another. This causes load imbalance and degrades the performance.

We found that a monitoring methodology requiring performance measurements on *all* core types is difficult to use in practice. Although asymmetry-aware scheduling algorithms show a strong potential to maximize performance on AMPs, how the algorithm is designed makes a big difference, since excessively heavy online monitoring can cause prohibitive overheads. Bringing to light the problems with seemingly simple and effective monitoring methodologies and proposing asymmetry-aware algorithms that are not susceptible to similar deficiencies are the key contributions

[1] Evaluating a real OS implementation on an asymmetric (or heterogeneous) processor where cores differ in pipeline microarchitecture would require us to use a full-system simulator (i.e., a simulator that boots a real operating system) that can also simulate asymmetric hardware. Despite availability of such simulators (e.g., COTSon [60]), they still run in the kilohertz range when accurate simulation is required, so performing extensive evaluation with a large number of long-running workloads would be challenging.

of this chapter.

In evaluating HASS, we were also interested in comparing it to a relatively simple asymmetry-aware algorithm that shares fast cores among the threads in a round-robin fashion. To that end, we have designed and implemented a Heterogeneity-Aware Fair Share (HAFS) algorithm that ensures that the total time spent by each thread on a given core type is proportional to the number of cores of that type in the system. Our study reveals important overheads associated with a real implementation.

We have implemented the algorithms (HASS-S, HASS-D, IPC-Driven and HAFS) in the OpenSolaris operating system and evaluated them on two real multicore platforms made asymmetric via CPU frequency scaling. We found that HASS-S improves performance by as much as 12.5% for diverse workloads (i.e., workloads where applications significantly differ from each other in their architectural properties) relative to a asymmetry-unaware scheduler. The IPC-Driven algorithm, in contrast, improved performance by at most 7% and often even caused performance degradation. HASS-D delivers performance gains of up to 12%.

We also observed that HASS did not do as well on systems with shared caches. HASS's model for estimating performance on different core types did not account for shared caches, and so the mapping of threads to cores that it performed was not always optimal. Nevertheless, HASS improved performance even in these difficult conditions, outperforming both IPC-Driven and HAFS, and never performing worse than the default scheduler.

The rest of the chapter is organized as follows. Section 4.1 describes the methodology for constructing architectural signatures. Section 4.2 describes the design and implementation of the evaluated algorithms. Section 4.3 analyzes the performance results. Section 4.4 discusses related work. Section 4.5 summarizes our findings.

4.1. Architectural signatures

An architectural signature is a summary of the architectural properties of an application. HASS relies on the ability to estimate the relative performance of threads on different core types. To that end, the signature must enable it to predict a thread's performance based on the features of the core. As explained earlier, we focus on systems where cores differ in clock frequency (as on the target evaluation platforms used in this thesis), a parameter expected to play a prominent role in future asymmetric systems.

To predict performance variations due to clock frequency, we must consider the application's degree of *memory intensity* [61]. As pointed out in Section 2.1.1, an application with a high rate of memory accesses is likely to stall the core often, so the clock frequency will not have a significant effect on performance. Memory intensity can be approximated by a thread's miss rate [62]. The static version of HASS estimates the miss rate from an application's reuse-distance profile, which is

obtained offline prior to executing the application. The dynamic version of HASS measures the miss rate online. The relative performance experienced by threads on cores with different frequencies is then estimated online by the scheduler using a simple performance model.

The static version is more appropriate for environments like embedded systems, where application inputs are typically known *a priori*, and so the architectural signatures can be obtained for all typical executions of the workload. The dynamic version is more appropriate for dynamic and highly phased workloads, whose runtime properties are too variable for capturing offline.

The remainder of this section is structured as follows. In Sections 4.1.1 and 4.1.2, we explain how the signatures are constructed for the static and dynamic version of HASS, respectively. Then, in Section 4.1.3, we explain how the scheduler estimates the relative performance of applications on different core types. Section 4.1.4 is devoted to describing how architectural signatures could be extended for multithreaded applications. In Section 4.1.5, we analyze the implications of the presence of shared caches in our signature-based framework and suggest how it can be improved for these scenarios.

4.1.1. Static signatures

Static signatures rely on a reuse-distance profile. A reuse-distance is defined as the number of intervening memory accesses between two consecutive accesses to the same memory location. A reuse distance profile is the distribution of reuse distances. From this profile we can accurately estimate the application's last-level cache miss rates for any cache configuration [59, 63, 64, 65] that can be encountered at runtime. *These estimated miss rates make up the contents of the static signature.*

Since reuse-distance profiles are mostly microarchitecture independent, our statically generated architectural signatures are microarchitecture independent as well[2]. Thanks to this property, out of all hardware platforms where it is possible to run the binary, we should be able to select any single one to construct a signature usable by all.

The signature should be available to the OS at scheduling time, so the ideal place to hold it is in the application binary itself. For evaluation covered in this chapter, however, we have not implemented the binary embedding scheme. Instead, we rely on a simple launcher program that injects the application's signature into the kernel right before the application starts. To make that possible, we augmented the OS with a new system call that enables us to communicate the signature of an application to the thread scheduler. The new system call is invoked by the launcher program right after switching to the application's code via `exec()`.

[2]LLC miss rates eventually depend on both reuse-distance profiles and specific microarchitectural features such as pre-fetching mechanisms, cache replacement policies an so forth. Nevertheless, for our purpose, i.e. guiding scheduling decisions on asymmetric systems, these statically generated signatures are enough to model the memory behavior of the applications.

To construct the signature, we need to obtain the reuse-distance profile, which is collected via offline profiling. Such profiling can be done, for example, as part of the feedback-directed optimization phase of the application development, which can be set up with little or no involvement from the programmer. All that needs to be done is to execute a program once with the profiler (see below) that will generate the signature and embed it into the binary. The responsibility of the developer, then, is to make sure that the thread exhibits "typical" behavior during this signature run. If it is impossible to do so in one run, the developer can do several runs (for example with different inputs) and combine the results into one signature. In this thesis, we constructed profiles using the method proposed by Shelepov et al. in [64]. Such profiles are extracted with Pin, a binary instrumentation framework from Intel [51], along with a custom extension to Pin, MICA (see more details in Section 3.5.1). Once the profile is collected, we estimate (also offline) cache misses for a limited set of realistic last-level cache configurations (we do not account for different first- and mid-level cache configurations, because we found that accounting for these details did not significantly affect the signatures' accuracy). These estimations, collected in a matrix, comprise the architectural signature. We support 11 different last-level cache sizes (powers of two from 16K to 16M) and four set associativities (4, 8, 16 and 32), so the matrix has 44 values.

Table 4.1: The architectural signature for `art`

set. assoc.	cache size						
	256K	512K	1M	2M	4M	8M	16M
4	427	412	325	157	42	6	1
8	427	418	332	131	21	1	0
16	427	424	337	107	11	0	0
32	427	426	336	86	5	0	0

Shown in Table 4.1 is an example signature for the benchmark `art` from the SPEC CPU2000 suite. (Columns for sizes 16K to 128K are omitted, because these values were in this case exactly the same as that for 256K (427 misses).) Each integer in the matrix cell represents the expected number of misses per 4096 instructions (the number 4096 was selected to speed up calculations at scheduling time).

When signatures are generated offline, capturing the differences between various phases is not impossible [66], but certainly more difficult[3]. Using multiple signatures for an application, representing its different program phases, may lead to improving the dynamic thread-to-core assignments further, but at the expense of extra complexity, due to phase detection, and potentially higher runtime overheads, due to additional thread migrations. We found that, in practice, the average behavior captured by a single signature is good enough to effectively guide scheduling decisions with low runtime overhead.

[3]Capturing the phased behavior of applications offline would require partitioning the program into phases that behave differently enough to have a different signature.

4.1.2. Dynamic signatures

Dynamic signatures are constructed quite simply by measuring the application's last-level cache miss rate online. The advantage of dynamic signatures relative to static ones is that they adapt to the variations in the program input and to program phase changes. Phase-change adaptability of the HASS-D algorithm is described in detail in Section 4.2.2. The disadvantage of the dynamic signature scheme is that it is not as easily adaptable to systems where different cores have different-sized last-level caches; on these systems a scheduler would need to run each thread on each core type, which, as will be shown later, degrades the performance. Therefore, another alternative for constructing dynamic signatures is to use dynamically estimated reuse-distance profiles (as in the system RapidMRC [67]), and as in the static case use these reuse-distance profiles to estimate the miss rates. Since our experimental systems had uniform cache sizes across cores, we relied on the first method, where the miss rates are measured directly online, rather than obtained via dynamically estimated reuse-distance profiles.

4.1.3. Using signatures for scheduling

At runtime the architectural signature is used to estimate a thread's performance on each type of core present in the system. To accomplish this, we calculate a hypothetical completion time for some constant number of instructions. Two separate components of completion time are considered: execution time and stall time. Execution time is the amount of time it takes to execute the instructions assuming a constant number of cycles per instruction. To compute the execution time we assume a cost of 1.5 cycles per instruction and factor in the clock speed. These two parameters are machine dependent so their values must be appropriately chosen for the hardware platform in question.

We approximate the stall time by the number of cycles used for servicing the last-level cache misses. Although this is a coarse approximation, it gives reasonable accuracy, because memory access time dominates other stalls [24]. To estimate this, we need memory access latency (discoverable by the OS) and the miss rate that we obtain from the signature. Note that since we are assuming a constant memory latency, the presence of non-uniform memory access (NUMA) can reduce the accuracy of estimates. Although we did not have a chance to investigate this effect comprehensively, we observed that in our case the presence of NUMA on one of the experimental platforms did not prevent the algorithm from performing successfully.

The resultant sum of both time components gives us an abstract "completion time" metric. For actual scheduling, we focus on the ratio of completion times calculated for different types of cores, to which we refer as the *Speedup Factor*. More precisely, if $P(t, C)$ denotes the performance for a given thread t on core C, and C_1, C_2 are two different core types such that C_1 is faster than C_2, then by convention we define the Speedup Factor as $SF(t, C_1, C_2) = \frac{P(t,C_2)}{P(t,C_1)}$.

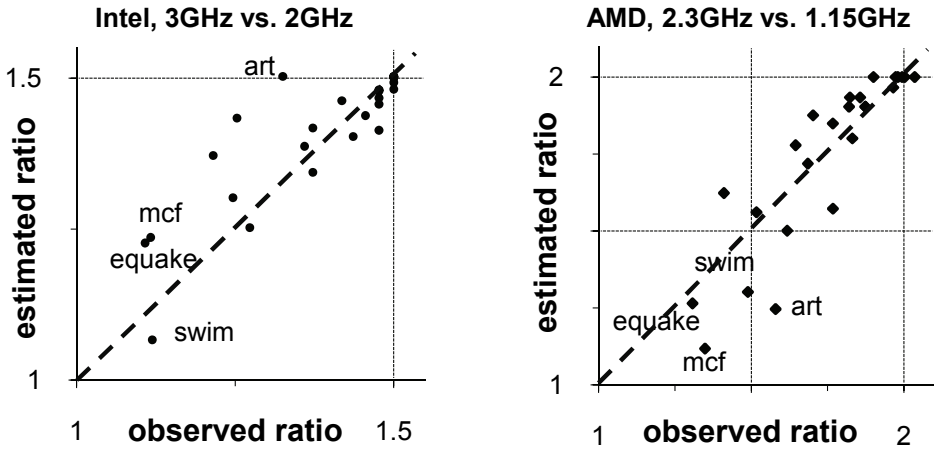


Figure 4.1: Signature-estimated performance ratios vs. observed ratios. Some outliers labeled. Perfectly accurate estimations would have all points on the diagonal line.

To summarize, we predict the performance of different threads on different cores based on threads' caching behavior and cores' frequency. This allows the OS to distinguish cores by their relative desirability for different threads. We have tested the accuracy of this method on cores that differ in frequency by using Dynamic Voltage and Frequency Scaling (DVFS) facilities available on most modern processors. DVFS allows the operating system to control the clock speed of the cores. Figure 4.1 shows how well real speedup factors match predicted speedup factors for some of our test configurations (described in Section 4.3). As evident, the estimation method is successful in separating memory-intensive threads (which are less sensitive to changes in frequency and therefore concentrated toward lower left) from CPU-bound threads (upper right), but is less precise in characterizing memory intensity.

4.1.4. Multithreaded applications

Although our signature-based framework was designed for single-threaded applications, there are no inherent barriers to extending it for multithreaded applications. In that case, the signature would be generated per thread – threads would be identified by the function that a thread executes. In scenarios where threads perform a different type of work (and thus have different architectural properties) despite executing the same function, an online method for signature generation would be preferred. The important point is that almost no changes would have to be done in the scheduler itself, because it already uses threads as schedulable entities associated with an architectural signature. This study, where for simplicity we use single-threaded applications in our experiments, evaluates the effectiveness on an asymmetry-aware scheduling algorithm assuming that per-thread signatures are known. Performing this evaluation was our key objective.

4.1.5. A reflection on shared caches

Wrapping up the discussion of architectural signatures, we would like to reflect on shared caches. On shared-cache architectures (including SMT), performance is affected not only by the frequency of the core and the properties of the application, but by cache access patterns of co-scheduled threads. Our existing method for estimating performance does not account for effects of shared caches. In our evaluation, this caused performance benefits to diminish when shared caches were present. Modeling shared cache effects is an orthogonal and well-studied problem. Existing models of miss rates in shared caches are based on input data very similar to reuse-distance profiles (used to construct our signatures) [68], and this presents a good opportunity to extend our signature-based model to account for cache sharing. For example, we can take advantage of different methods to perform optimal co-scheduling of threads on shared-cache architectures based on the threads' reuse-distance profiles, such as the algorithm presented in [62]. That algorithm is able to find the optimal thread schedule most of the time, performing within 1% of the oracular algorithm that always picks the optimal assignment. Integrating this cache-aware scheduling algorithm with asymmetry-aware algorithms is an interesting avenue for future work.

4.2. The algorithms

In this section we describe the scheduling algorithms evaluated in this chapter and highlight the main challenges we had to face when creating real-world implementations of these. The investigated scheduling algorithms follow different rules when assigning threads to fast and slow cores, but they all rely on the design approach to asymmetry-aware scheduling described in Section 2.3.2 (based on the *core partition* abstraction). Therefore, all the cores in the system are organized into *fast* and *slow partitions* – non-overlapping sets of cores consisting of fast and slow cores, respectively; the asymmetry-aware schedulers perform thread assignments to core partitions (sets of cores) rather than to individual cores.

In Sections 4.2.1 and 4.2.2, we describe the static and dynamic versions of HASS, respectively. Section 4.2.3 focuses on the IPC-Driven algorithm. Section 4.2.4 outlines the HAFS algorithm.

4.2.1. The HASS-S algorithm

A key goal in the design of HASS was scalability, because future multicore processors may be built with hundreds or thousands of cores. Scalability mainly manifests in two aspects of the algorithm design: the lack of global locks, and the scheduling decision logic that relies only on local information. As we describe the algorithm, we will point out the particular features that ensure scalability.

The HASS-S algorithm relies on core partitions. The scheduler maintains a counter of runnable threads (threads either currently running or ready to be run) for each core partition. This counter is the primary partition-wide contention point, as it has to be fully synchronized. In HASS-S, a partition is the widest locking scope during normal operation of the scheduler. This partition-based design enables us to manage a large number of cores in a scalable way.

When threads enter the system, the operating system estimates their performance on fast and slow cores according to the attributes of both core types (following the method described in Section 4.1.3). The ratio between these estimates is the aforementioned speedup factor, which approximates the fast-to-slow speedup that a thread would experience when running on a fast core without ever being preempted in favor of other threads.

Algorithm 4.1 An algorithm for *regular assignment* and *optimistic rebinding* in HASS-S

Definitions: F is the set of threads assigned to fast cores, S is the set of threads assigned to slow cores. FP and SP is the set of fast and slow partitions respectively. t is a runnable thread.
Require: $(t \in F \bigcup S)$
Ensure: t is assigned to a partition that improves its performance.
  success $\leftarrow$ **false**
  { First of all, try a *regular assignment* }
  $p_{cur} \leftarrow$ partition where t is currently assigned to
  if $((t \in F)$ **and** $(nthreads(p_{cur}) \leq ncores(p_{cur}))$ **then**
    { The expected performance of t is already optimal }
    success $\leftarrow$ **true**
  else { Find a better target partition to improve performance }
    $p_{target} \leftarrow$ find partition $p_s \in (FP \bigcup SP)$ with maximum expected performance for t
    if $expected\_performance(t, p_{target}) > expected\_performance(t, p_{cur})$ **then**
      move t to p_{target}
      success $\leftarrow$ **true**
    end if
  end if
  { Try *optimistic rebinding* if no better partition was found so far }
  if not *success* **then**
    $t_{partner} \leftarrow$ find thread $t_{partner}$ such that the swap of t with $t_{partner}$ improves system performance
    if $t_{partner}$ was found **then**
      swap t and $t_{partner}$
    end if
  end if

To assign a thread to a specific partition, the scheduler goes through the list of all partitions and estimates that thread's performance in each partition using the speedup factor and the current number of runnable threads per core in it. The

scheduler assumes that the CPU time will be shared equally among all threads within the partition. After that, the scheduler selects the partition with the highest expected performance and assigns the thread there. This process is called *regular assignment*. Note that regular assignment has linear complexity with respect to the number of partitions, so there should be a balance between the number of partitions and the number of cores in each partition. An assignment of threads to partitions is not only done initially, but is repeated every time a thread accumulates a certain amount of CPU time on its current partition, in case the current partition becomes non-optimal, i.e., when the number of threads in a partition changes. This repeated assignment is called a *refresh*. By having the refresh period tied to CPU time rather than wall clock time, we avoid increasing the absolute number of refreshes as the load factor grows. If there is no change in the system load, the refresh assignment can be skipped.

Algorithm 4.1 shows the pseudo-code for the regular assignment and refresh. *Optimistic rebinding*, also shown in this listing is discussed in the following paragraphs. For reasons of conciseness we do not provide the pseudo-code for *expected_performance*. The procedure for estimating the performance on different core types is described in Section 4.1.3.

The greedy thread assignment approach described so far has a potential problem where threads may become locked in a suboptimal assignment and further optimization can only be accomplished by cooperative action between two threads (swapping) rather than by a greedy decision w.r.t. one thread. There is a mechanism to perform such swapping, and it is called *optimistic rebinding*. A scheduler may decide to use optimistic rebinding instead of the regular assignment during a refresh, if it fails to find a good target partition for a thread. The scheduler then has to find a partner for the thread in some partition with "good" potential performance and swap the target thread with the partner. The scheduler only triggers the swap if it confirms that the swap will actually increase the performance of the target thread as well as the overall system performance. This is done by comparing the speedup factors of the target thread and of the potential partner. The search for a partner can be slow when the target partition has many threads or when there are a lot of partitions. Therefore, our algorithm forgoes exhaustive search and instead uses randomized search with a limited amount of probing.

The partitioning scheme allows the scheduler to avoid global synchronization during scheduling. Instead, threads can lock one partition at a time when doing a refresh (for reading the runnable threads counter), migrating between partitions or entering/leaving runnable states (for updating the runnable counter). Using read/write locks can further decrease the pressure on this contention point.

4.2.2. The HASS-D algorithm

HASS-D, the dynamic version of HASS-S, estimates the speedup factor online, by periodically sampling threads' last-level-cache (LLC) miss rates and using them as the input to the performance algorithm described in Section 4.1.3. The fact that the

Algorithm 4.2 Event-driven migrations in HASS-D

Definitions: F is the set of threads assigned to fast cores, S is the set of threads assigned to slow cores, t is a runnable thread whose SF has changed.

Require: $(F \neq \emptyset) \wedge (S \neq \emptyset) \wedge (t \in F \bigcup S) \wedge$
$\wedge (t \in F \Rightarrow (\forall u \in F - \{t\}, \forall v \in S : SF(u) \geq SF(v))) \quad \wedge$
$(t \in S \Rightarrow (\forall u \in F, \forall v \in S - \{t\} : SF(u) \geq SF(v)))$

Ensure: $(\forall u \in F, \forall v \in S : SF(u) \geq SF(v))$

```
if t ∈ F then
    t_sc ← find thread t_sc in S with max SF
    if SF(t) < SF(t_sc) then
        {swap threads t and t_sc}
        < F, S > ← < F − {t} + {t_sc}, S − {t_sc} + {t} >
    end if
else {t ∈ S}
    t_fc ← find thread t_fc in F with min SF
    if SF(t) > SF(t_fc) then
        {swap threads t and t_fc}
        < F, S > ← < F − {t_fc} + {t}, S − {t} + {t_fc} >
    end if
end if
```

speedup factor in HASS-D is not known when the thread first arrives, and that it can change dynamically throughout the thread's lifetime, dictates different algorithms for assignment of threads to cores than those used in HASS-S. For example, HASS-D cannot perform the same regular assignment as HASS-S, because when the thread arrives its speedup factor is not yet known. Likewise, subsequent reassignment of threads in HASS-D is driven by dynamic changes in the speedup factor as well as by changes in the load. The main focus of this section, therefore, is to describe the algorithm used in HASS-D to assign threads to cores.

When a new thread enters the system, HASS-D assigns it a *default* speedup factor[4], since no information about its actual speedup factor is available. The initial mapping of newly created threads is performed such that fast partitions are populated before slow partitions and the load balance across the cores is preserved (see Section 2.3.2 for more details on our approach to load balancing). As soon as the thread begins to run, HASS-D begins to monitor its last-level cache miss rate (on whatever core it was assigned to run), and then uses that miss rate to estimate its speedup factor as described in Section 4.1.3.

As threads run, two things can happen: speedup factors for newly arrived threads become known, or speedup factors of old threads (as they enter into different phases of execution) change. The scheduler must map threads to cores according to their speedup factors, and to that end it follows the so-called *event-driven migration*

[4]For this default value, we opted to choose the lowest speedup factor attainable in an attempt to avoid that threads with a relatively low estimated speedup factor and legitimately assigned to fast cores are evicted from fast partitions when new threads enter the system.

procedure shown in Algorithm 4.2 and described below.

Event-driven migrations ensure that the system adheres to the following two rules: (1) All threads in fast partitions have a higher *SF* than the thread with maximum *SF* running in a slow partition (2) load balance must be preserved.

In order to enforce Rule 1, the scheduler must check that the thread with minimum *SF* on fast cores (t_{fc}) has a higher *SF* than the thread with highest *SF* on slow cores (t_{sc}). This rule may be broken either when a change in the *SF* of a thread takes place or in the event that a thread transitions between a runnable and a non-runnable state. In the former scenario, the scheduler enforces the rule by swapping t_{fc} and t_{sc} when needed. In the latter case, the migration of one the aforementioned threads when necessary is enough to guarantee that the two rules of HASS-D hold true.

The scheduler maintains per-partition lists of runnable threads sorted by *SF* to simplify the selection of the optimal thread(s) to migrate (or swap): t_{fc} and t_{sc}. For efficiency reasons, fast partitions' lists are kept sorted in an ascending order by *SF*, while a descending order by *SF* is preferred for thread lists in slow partitions. As a result, finding the optimal candidate in either case has linear complexity with respect to the number of partitions.

HASS-D measures LLC miss rates for each thread continuously using performance counters, and the values are sampled every 20 timer ticks (roughly 200ms on our experimental system). We keep a running average of the values observed at different periods and we discard the first values collected immediately after the thread starts or after it is migrated to another core in order to correct for cold-start effects causing the miss rate to spike intermittently after migration.

We also use a phase-detection mechanism that seeks to capture coarse-grained phases rather than fine-grained ones, in an attempt to reduce the number of unnecessary migrations. Updating *SF* estimations during abrupt phase changes may cause frequent expensive migrations, which may end up being unnecessary if the new phase is too short. Instead, *SF* estimations are updated exclusively once a thread enters a phase exhibiting stable behavior.

To detect stable phases, we use a light-weight mechanism based on a `phase_transition_threshold` parameter (12% in our experimental platform). When the running average is recorded, it is compared with the previous average measured over the previous interval. If the two differ by more than the transition threshold, a phase transition is indicated. Two or more sampling intervals containing no indicated phase transition signal a stable phase.

4.2.3. The IPC-Driven algorithm

To compare HASS to an existing asymmetry-aware algorithm, we chose to implement the IPC-Driven algorithm proposed previously by Becchi and Crowley [2], an algorithm that combined good results, applicability to general purpose systems and

Algorithm 4.3 IPC-Driven's thread swapping mechanism

Definitions: F and S are the sets of threads assigned to fast and slow cores, respectively. F_p and S_p are the sets of *pinned* threads on fast and slow cores, respectively. t is a runnable thread whose *IPC-ratio* has changed or a thread that has just entered the *pinned* state.

Require: $(F_p \subseteq F) \wedge (S_p \subseteq S) \wedge (F_p \neq \emptyset) \wedge (S_p \neq \emptyset) \wedge (t \in F_p \bigcup S_p) \wedge$
$\wedge (t \in F_p \Rightarrow (\forall u \in F_p - \{t\}, \forall v \in S_p : \textit{IPC-ratio}(u) \geq \textit{IPC-ratio}(v))) \wedge$
$\wedge (t \in S_p \Rightarrow (\forall u \in F_p, \forall v \in S_p - \{t\} : \textit{IPC-ratio}(u) \geq \textit{IPC-ratio}(v)))$

Ensure: $(\forall u \in F_p, \forall v \in S_p : \textit{IPC-ratio}(u) \geq \textit{IPC-ratio}(v))$

if $t \in F_p$ **then**
  $t_{sc} \leftarrow$ find thread t_{sc} in S_p with max *IPC-ratio*
  if $\textit{IPC-ratio}(t) < \textit{IPC-ratio}(t_{sc})$ **then**
    {swap threads t and t_{sc}}
    $< F, S, F_p, S_p > \leftarrow < F - \{t\} + \{t_{sc}\}, S - \{t_{sc}\} + \{t\}, F_p - \{t\} + \{t_{sc}\}, S_p - \{t_{sc}\} + \{t\} >$
  end if
else $\{t \in S_p\}$
  $t_{fc} \leftarrow$ find thread t_{fc} in F_p with min *IPC-ratio*
  if $\textit{IPC-ratio}(t) > \textit{IPC-ratio}(t_{fc})$ **then**
    {swap threads t and t_{fc}}
    $< F, S, F_p, S_p > \leftarrow < F - \{t_{fc}\} + \{t\}, S - \{t\} + \{t_{fc}\}, F_p - \{t_{fc}\} + \{t\}, S_p - \{t\} + \{t_{fc}\} >$
  end if
end if

specification completeness. In the original work [2] the IPC-Driven scheduler was simulated. We created the first real implementation of the IPC-Driven algorithm.

The IPC-Driven algorithm assumes two types of cores: ("fast") and ("slow"). The assignment is done based on IPC ratios, which determine the relative benefit of running a thread on a particular core type. IPC ratios in the IPC-driven algorithm are synonymous with the speedup factor in the HASS algorithm, but in this section we will use the term IPC ratio to follow the original definition of the authors.

The key idea behind the IPC-Driven algorithm is very similar to HASS-D: IPC-Driven like HASS-D also relies on event-driven migrations, and the procedures for event-driven migrations in the two algorithms are very similar. The key difference is that IPC-Driven *requires running each thread on both core types* to estimate its IPC ratio, while HASS-D only needs to run a thread on one (any) core type to estimate its speedup factor. As we will see later, the need to run a thread on both core types creates load imbalance, which causes performance degradation, and often results in inaccurate estimates of the IPC ratios. We will provide more discussion and explanation of this phenomenon in the experimental section. In the rest of this section, we complete the description of the IPC-Driven algorithm.

A thread with a high ratio between the IPC on the fast core and the IPC on the slow core is expected to benefit from the fast core. The scheduler periodically samples

threads' IPC on both core types and examines the IPC ratios of threads running on fast and slow cores. If the smallest IPC ratio among the threads running on the fast core is smaller than the highest IPC ratio among the threads running on the slow cores, the threads with the corresponding ratios are swapped. This part of the IPC-Driven algorithm is very similar to the event-driven migration algorithm in HASS-D. It is shown in Algorithm 4.3.

Just like HASS-D, IPC-Driven periodically re-estimates the IPC ratio when a thread is deemed to have entered a new phase. New program phases are detected by the changes in the program's IPC that exceed a certain `ipc_threshold`. Whenever a program enters a new IPC phase, the IPC ratios relative to the thread's most recent "home" core type are re-measured. However, unlike HASS-D, re-estimating the IPC ratio requires migrating a thread to another core (and as we show in the experimental section, this is the main cause for performance differences between HASS-D and IPC-Driven). This is done via forced migrations where a thread is switched to run in a partition of the opposite core type to its most recent one for a period of time called `refresh_period`. Additionally, a forced migration is triggered for threads that have just entered the system (after a warm-up period) in order to initialize the IPC ratio. Note also that those newly created threads are assigned in the first place to the partition with the lowest number of runnable threads per core.

In order to limit the number of forced migrations and to allow the system to stabilize between two consecutive thread swaps, a thread must run on a new core for a period of time equal to a `swap_inactivity` period before another forced migration is allowed. A thread that has been assigned to a particular core and is eligible for swapping is said to be in a *pinned state*. A thread whose IPC ratio is in the process of being updated is said to be *refreshing*. The performance of the IPC-Driven algorithm is sensitive to the settings of the aforementioned parameters (`refresh_period`, `swap_inactivity period`, etc.), and so we have carried out an exhaustive evaluation of the parameter space and picked the ones that yielded the best overall performance. `Refresh_period` was set to 30 milliseconds, `ipc_threshold` to 10%, `swap_inactivity` period to 1.5 seconds and `warm_up` period to 200 milliseconds.

4.2.4. The HAFS algorithm

HAFS is an implementation of an asymmetry-aware round-robin scheduling policy. The goal of this algorithm is to ensure that fast cores are shared equally among threads. The current implementation supports systems with two types of cores: fast and slow.

On the high level, the HAFS algorithm works as follows. It assigns threads to slow and fast partitions so as to preserve load balance across the cores, and then periodically migrates the threads among fast and slow partitions to ensure that fast cores are shared equally among the threads. HAFS relies on two mechanisms: *Inter-partition swaps* and *balance counters*. Inter-partition swaps is a mechanism for cross-partition migrations that ensures that migrations do not disturb load balance.

Algorithm 4.4 HAFS's thread swapping mechanism

Definitions: F and S are the sets of threads assigned to fast and slow cores, respectively. F_x and S_x are the sets of *expired* threads on fast and slow cores, respectively. t is a runnable thread that has just entered the *expired* state. BC stands for *balance counter*.

Require: $(F_x \subseteq F) \wedge (S_x \subseteq S) \wedge (F_x \neq \emptyset) \wedge (S_x \neq \emptyset) \wedge (t \in F_x \bigcup S_x) \wedge (((t \in F_x \Rightarrow (F_x = \{t\})) \wedge (t \in S_x \Rightarrow (S_x = \{t\}))$

Ensure: $((F_x = \emptyset) \vee (S_x = \emptyset)) \wedge$
$\wedge$ t has been swapped with t_x, the *oldest* expired thread in the opposite core type

if $t \in F_x$ **then**
  $t_x \leftarrow$ find thread t_x in S_x with max BC
  {swap threads t and t_x}
  $< F, S, F_x, S_x > \leftarrow < F - \{t\} + \{t_x\}, S - \{t_x\} + \{t\}, F_x - \{t\}, S_x - \{t_x\} >$
else $\{t \in S_x\}$
  $t_x \leftarrow$ find thread t_x in F_x with min BC
  {swap threads t and t_x}
  $< F, S, F_x, S_x > \leftarrow < F - \{t_x\} + \{t\}, S - \{t\} + \{t_x\}, F_x - \{t_x\}, S_x - \{t\} >$
end if

Balance counters is a mechanism that ensures that fast cores are shared equally among the threads. We first explain how inter-partition swaps work, and then describe the balance counters.

Suppose that a thread must be migrated from one partition to another. Simply enqueueing this thread in a runqueue of a core in the target partition could cause load imbalance if there is a large number of migrations going one way. To prevent load imbalance, the scheduler never migrates a thread from one partition to another unless there is a *candidate* thread that needs to be migrated in the opposite direction. *Swapping* threads among partitions, rather than performing one-way migrations, is guaranteed to preserve load balance.

A thread may be migrated without a swap if there are idle cores in a fast partition, since one goal of HAFS is to keep the fast cores busy. Furthermore, if a fast partition is overloaded and there are idle cores in a slow partition, the algorithm will also migrate a thread into that slow partition without requiring a swap. This preserves load balance.

Balance counters are used to achieve fair sharing. The scheduler associates with each thread a "balance" counter to track the deviation between the number of cycles a thread has been running in slow partitions compared to fast partitions. When that counter reaches a certain threshold[5], the scheduler sets the thread as "expired" and marks it as a candidate for migration onto the opposite core type. When a matching candidate thread wishing to migrate in the other direction appears, the two threads are swapped. Candidates are swapped in the FIFO order, so no thread gets "stuck"

[5]There is actually a positive and a negative threshold. The former controls the number of cycles a thread should spent in slow partitions without being migrated, whereas the latter performs the same control in fast partitions.

in a slow partition longer than any other thread. The swapping mechanism, which ensures that fast cores are shared equally, is illustrated in Algorithm 4.4.

If there are no matching candidates for swapping, an "expired" thread will keep running in the old partition. For example, if the number of threads is smaller than or equal to the number of fast cores, all the threads will keep running on fast cores without ever being migrated to slow cores.

Inter-partition swaps and balance counters ensure that fast cores are shared equally among threads and that the load balance is preserved at the same time. Another important property of HAFS is that it does not require global communication across all cores when making scheduling decisions. This property of the algorithm suggests that it has good scalability properties, which will be especially relevant on future many-core systems with potentially hundreds of cores.

4.3. Results and discussion

The algorithms presented so far were implemented as separate scheduling modules in the OpenSolaris operating system, following the implementation approach described in Section 3.4.4. In this section, we report on our experience in evaluating these algorithm on two real multicore platforms made asymmetric via dynamic voltage and frequency scaling.

This section is divided into four parts. In Section 4.3.1 we introduce the investigated asymmetric configurations. In Section 4.3.2, we enumerate the benchmarks and workloads used for the evaluation. Section 4.3.3 outlines our experimental methodology. Finally, in Section 4.3.4, we analyze the performance results of all the investigated schedulers and report our main findings.

4.3.1. Asymmetric configurations

Our evaluation was carried out on the Intel-8 and the AMD-16 platforms presented in Section 3.3. We configured our test systems to be asymmetric by setting the frequency of fast and slow cores to the maximum and minimum frequency levels attainable, respectively. In the asymmetric configurations based on the Intel-8 platform, fast cores operate at 3.0 GHz, while slow cores run at 2.0 GHz. Conversely, in the AMD-16 platform, fast and slow cores were set to run at 2.3 GHz and 1.15 GHz, respectively.

In our experiments we used three asymmetric configurations: (1) 2FC-2SC-A – based on AMD-16 with two fast cores and two slow cores, each on its own chip and exclusive L3$ per core; (2) 2FC-2SC-B – based on Intel-8 with two fast cores and two slow cores, each on its own chip and exclusive L2$ per core; and (3) 4FC-12SC – four fast cores and twelve slow cores on the AMD-16 platform. In order to avoid any performance effects due to cache sharing in 2FC-2SC-A and 2FC-2SC-B, we used fewer cores than available in the machine (to that end, we had to use at most

Table 4.2: Multi-application workloads (a) Set #1, (b) Set #2

(a) Workload set #1

Categories	**Benchmarks**
HH1	sixtrack, crafty, mcf, equake
HH2	gzip, sixtrack, mcf, swim
HH3	mesa, perlbmk, equake, swim
LH1	wupwise, wupwise, wupwise, wupwise
MH1	vortex, twolf, fma3d, art
MH2	gap, parser, applu, vpr
MH3	apsi, ammp, lucas, mgrid
MH4	bzip2, gcc, wupwise, art

(b) Workload set #2

Categories	**Benchmarks**
W1	sixtrack, crafty, eon, gzip, twolf, mesa, parser, bzip2, gap, vortex, ammp, mgrid, gcc, apsi, vpr, wupwise
W2	sixtrack, crafty, twolf, perlbmk, mesa, parser, bzip2, gap, vortex, ammp, mgrid, apsi, vpr, wupwise, fma3d, art
W3	gzip, bzip2, parser, gap, vortex, ammp, mgrid, gcc, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas
W4	eon, gzip, perlbmk, gap, mgrid, gcc, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W5	gzip, sixtrack, crafty, perlbmk, gap, mgrid, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W6	parser, gap, vortex, ammp, mgrid, gcc, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W7	sixtrack, crafty, twolf, mesa, gap, mgrid, apsi, vpr, wupwise, fma3d, art, applu, swim, lucas, mcf, equake
W8	sixtrack(x2), crafty(x2), art(x2), applu(x2), swim(x2), lucas(x2), mcf(x2), equake(x2)

one core per chip). Conversely, the 4FC-12SC configuration, where all of the cores are used, is subject to cache interference effects.

4.3.2. Benchmarks

Our workloads consist of several single-threaded applications drawn from the SPEC CPU2000 suite that expose a wide range of behaviors concerning the efficiency of pipeline utilization. Although reuse-distance profiles for HASS-S had to be collected on Linux (Pin does not run on OpenSolaris), we ensured that the benchmark binaries compiled for Linux-x86 were sufficiently similar to the binaries compiled for Solaris-x86 by using the same compiler version and flags.

We opted not to include multithreaded applications in our workloads because our investigated algorithms only seek to deliver ILP specialization rather than TLP (thread-level parallelism) specialization. As we will see in Chapter 6, the algorithms whose main goal is to deliver TLP specialization only, such as “PA” (described in Chapter 5), turn out beneficial for workloads containing both parallel and sequential

applications but are unable to deliver performance gains for workloads consisting of single-threaded applications only.

In our experimental evaluation we used two workload sets: #1 and #2. Benchmarks included in each set are shown in Tables 4.2a and 4.2b. Workload set #1 includes eight workloads with four applications each. Workload set #2 consists of eight application sets, each including up to sixteen different benchmark programs.

Set #1 includes three categories of workloads. The first category is highly heterogeneous (HH), and consists of a pair of highly CPU-bound benchmarks and a pair of memory-bound benchmarks. Workloads in this category (HH1, HH2 and HH3) show the most diversity in terms of applications' architectural properties, and so they will have the highest performance improvements from asymmetry-aware algorithms. In each HH workload, the first two benchmarks are CPU intensive with virtually any cache size, and the second pair is memory intensive. These workloads, especially when running on the 2FC-2SC-A and 2FC-2SC-B configurations, enable us to assess the effectiveness of each investigated scheduler under the most favorable (most heterogeneous) conditions, since in those configurations an effective scheduling will result in mapping the two CPU-intensive instruction streams on the two available fast cores. The second category of workloads included in set #1 is moderately heterogeneous (MH). These workloads include benchmarks representing the whole spectrum of memory intensity, with less extreme differences between the benchmarks. In general, however, the first two benchmarks in each workload are less memory intensive than the last two on the majority of our configurations. These MH workloads are expected to benefit less from asymmetry-aware scheduling than HH workloads. Finally, in the lightly heterogeneous category (LH) we have the workload consisting of the four copies of the same application (`wupwise`). We report the data on only one workload in this category, because we did not observe particularly interesting effects for homogeneous workloads.

In all experiments, the total number of applications was set to match the number of cores in the asymmetric platform, since this is how runtime systems typically configure the number of threads when only CPU-bound applications are used [26]. Four copies of each benchmark from set #1's "base" workloads were used on the 4FC-12SC configuration to make use of the sixteen cores available, while only one copy is needed on the 2FC-2SC-B and 2FC-2SC-A configurations to keep all cores busy.

For the 4FC-12SC configuration, we also report the performance of workloads in set #2, which are larger and more diverse than those in set #1. Workloads in set #2, which include benchmark programs with a wide range of speedup factors, are displayed in Table 4.2b, where these appear sorted in ascending order by memory intensity. This way, the workloads range from W1 (the least memory intensive) which is made up of both CPU-intensive and mildly memory-intensive applications, to W8 (the most memory-intensive workload), which contains up to twelve highly memory-intensive programs.

4.3.3. Experimental methodology and metrics

For a given test we launch a predetermined number of benchmarks, and as individual copies terminate, they are immediately restarted. Thus we keep the workload constant and measure the average completion time of every benchmark. Our goal is to minimize the mean of normalized completion times across all benchmarks. Each benchmark runs at least three times, so there are at least three completion time values.

Our original goal was to compare the completion times achieved with HASS to the completion times achieved with the native OpenSolaris scheduler, but we found that completion times under the native scheduler were highly variable (standard deviation was as high as 23%) and thus not suitable for comparison. As explained in Section 2.3, this is due to the fact that the native scheduler is not asymmetry aware and thus migrates threads between different core types at infrequent and arbitrary intervals. Therefore, the fraction of time that a thread spends on a particular core type varies significantly from one run to another. Achieving a low standard deviation is not possible in these conditions.

Instead we compare completion times of the algorithms to a composite metric, to which we refer to as the *default metric*. To compute a completion time for a benchmark using the default metric we run the benchmark bound to a specific type of core (e.g., the "fast" type) while the rest of the benchmarks in the workload are running on other cores. Then we repeat the same measurement while the benchmark is bound to the core of the other type (e.g., "slow"). We then average the completion times on fast and slow cores, and the resulting value is used to approximate the default completion time. This metric gives us the expected completion time of a benchmark over a large number of trials if the benchmark were randomly bound to a core at the start of its execution and kept running on that core until completion. This is a good approximation of how the default scheduler operates, because it tries to minimize migration of threads from one core to another in order to maintain cache affinity.

The default metric could be too pessimistic on systems with sustained loads, where new threads are constantly arriving as old threads finish. As threads running on faster cores retire more often, faster cores will be available for assignment more often. To compensate, we also compare our algorithms to HAFS, which keeps the fast cores busy. It is important to understand though, that the performance achieved with HAFS will be better than with a asymmetry-agnostic default scheduler, because HAFS is specifically designed to keep the fast cores busy. In summary, while we do not use real completion times for the native scheduler, we understand that they are no worse than the default metric, and are somewhat worse than those obtained with HAFS.

For performance comparison, we report completion times normalized to the default metric for each benchmark in workload set #1 as well as the geometric mean for all its benchmarks. For the sake of clarity, however, we only show the geometric mean of normalized completion times for all benchmarks in workload set #2.

In addition to the results obtained with the various algorithms, we also show the results obtained with the best static assignment. A static assignment is decided at the beginning of execution and never changed thereafter. The best static assignment is obtained by testing all possible static assignments and picking the one with the best performance. The best static assignment is the theoretical upper bound for the performance that can be achieved with our implementation of HASS-S.

4.3.4. Performance analysis

Evaluation of HASS-S

First we analyze the behavior of HASS-S on the 2FC-2SC-A configuration. This is the configuration where we expect to see the best results, because (1) this configuration is more "heterogeneous" than the 2FC-2SC-B configuration, since the difference in the speeds of fast and slow cores is greater than on 2FC-2SC-B, and (2) this configuration is not subject to cache sharing (unlike the 4FC-12SC system), which our algorithms do not handle. Figures 4.2a and 4.2b show the completion times for the different workloads relative to the default metric (lower numbers are better). The types of workloads are shown on the bottom of the chart. The completion times are shown for each application individually as well as for the entire workload as the geometric mean.

As can be expected, HASS-S performed especially well with the HH workloads, where the mean speedup was as much as 12.5% for the {`sixtrack`, `crafty`, `mcf`, `equake`} workload and reached 10% and 11% for the other HH workloads. In all these cases HASS-S achieved its theoretical upper bound. We traced the execution of the benchmarks with DTrace and confirmed that HASS-S actually chooses the mapping of threads to cores that corresponds to the best static assignment.

As expected, the performance improvements were more modest for the MH workloads: 10%, 8%, 6% and 8% for each of the four workloads and 8% on average on the 2FC-2SC-A system. The reason is that there are smaller differences in the CPU speed sensitivities among different applications, so the optimization opportunities are smaller. Despite the similarity in the applications' signatures in the MH workload, HASS-S was still able to pick the right candidates for running on the fast cores, matching again the best static assignment.

To understand the source of performance improvements from asymmetry-aware scheduling, we examine the relative completion times for individual benchmarks within the workload. For HH and MH workloads we see that the first two applications in the workload (recall that these are CPU-bound applications) usually speed up under asymmetry-aware scheduling (i.e., they experience lower completion times), while the second two applications (the memory-bound type) slow down. Since the speedup experienced by the CPU-bound applications is greater than the slow-down experienced by the memory-bound applications, the workload as a whole experiences an improvement in performance. This points to the inherently "discriminative" nature of asymmetry-aware scheduling, which may make it inappropriate to

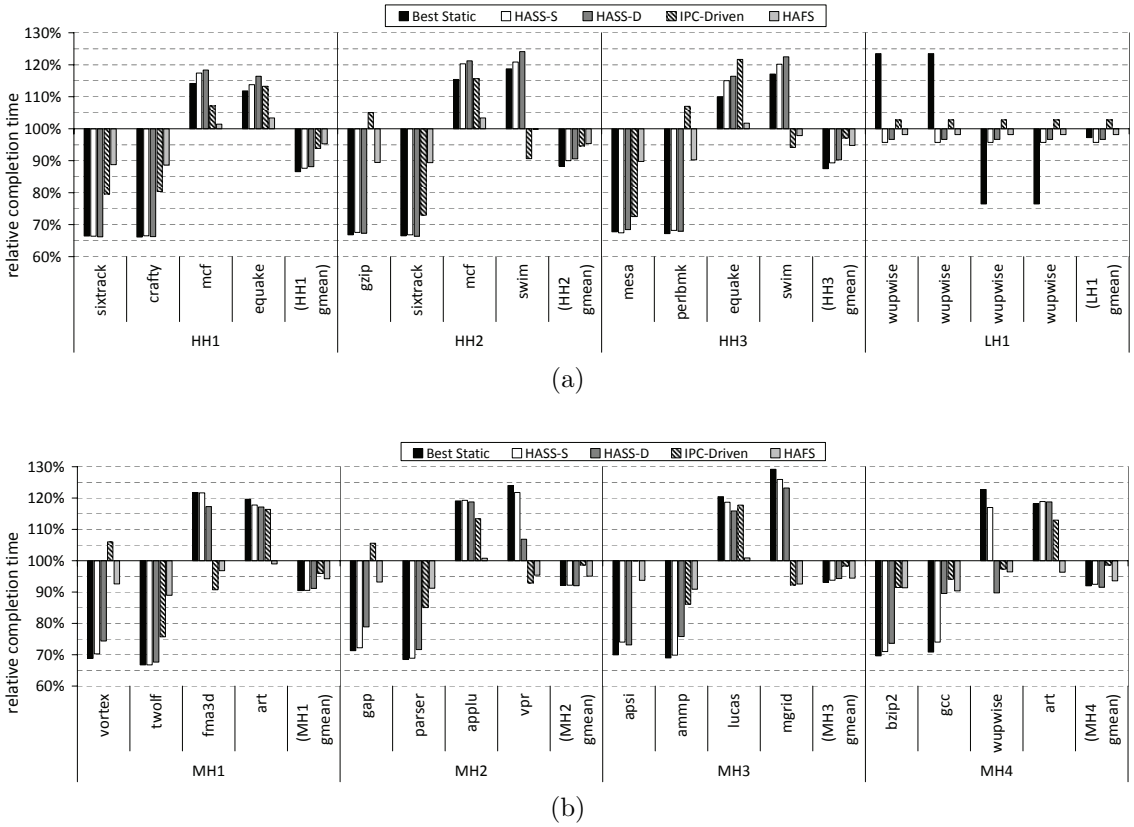


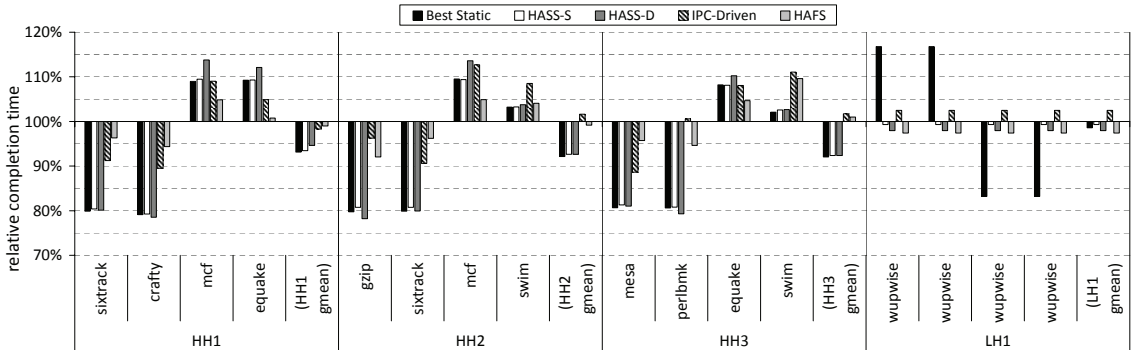
Figure 4.2: Completion times relative to the default metric for workload set #1 on the 2FC-2SC-A configuration. Bars above 100% represent slow down, and below 100% represent speedup. (a) HH and LH workloads, (b) MH workloads.

situations where the goal is to optimize the performance of individual applications. But when the goal is to optimize the workload as a whole, the asymmetry-aware policy does its job.

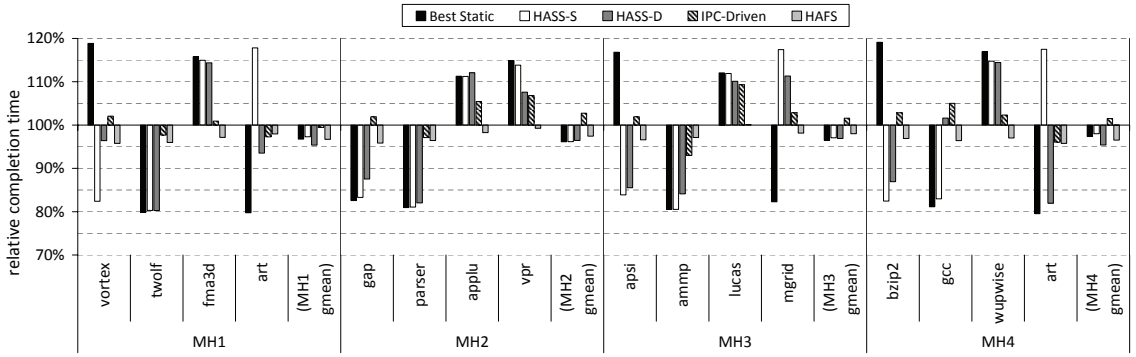
Examining completion times for individual applications offers another way to check whether HASS-S was able to pick the right candidates to run on the fast cores. Ideally, we want to see the same applications experiencing the speedup with HASS-S as with the best static assignment. For HH and MH workloads we see that this is always the case. HASS-S is able to determine which two applications are CPU intensive and assign them to run on fast cores, matching the best static assignment.

The speedup that HASS-S achieves relative to HAFS is more modest: on average 6% for HH workloads and 2% for MH workloads. This implies that for MH workloads, a simple asymmetry-aware round-robin scheduler can perform almost as well as more complex algorithms, but for HH workloads a more sophisticated assignment policy is necessary to optimize performance.

We also note that HASS-S always outperforms the IPC-Driven algorithm. This was a surprising finding, because the IPC-Driven algorithm, in contrast to HASS-S, is phase aware and so it could fine tune the thread assignment as the workload goes through different phases of execution. We provide the explanation for this unexpected result in Section 4.3.4.



(a)



(b)

Figure 4.3: Completion times relative to the default metric for workload set #1 on the 2FC-2SC-B configuration. (a) HH and LH workloads, (b) MH workloads.

The other investigated phase-aware scheduler, HASS-D, is not subjected to the same limitations that IPC-Driven suffers from, since performance monitoring in this scheduler does not require cross-core migrations. As a result, HASS-D performs within 1% range of HASS-S on the 2FC-2SC-A configuration.

Turning to the 2FC-2SC-B configuration (Figures 4.3a and 4.3b), we note that the range of performance improvements from asymmetry-aware scheduling is smaller on this system. This is expected, because this hardware platform is less "heterogeneous" than the 2FC-2SC-A configuration. This indicates that sophisticated asymmetry-aware algorithms are more appropriate for systems with a high degree of heterogeneity among the cores as opposed to systems where performance differences across the cores are small[6].

On the 2FC-2SC-B configuration, most benchmarks exhibited a more CPU-bound nature than on the AMD system (probably because the Intel system had larger L2 caches), and so there was less distinction in the CPU speed sensitivities among the benchmarks, especially those in the MH category. As a result, HASS-S often picked a different set of applications to run on fast cores than those picked by the best static assignment. Nevertheless, the differences in the sensitivities were so small

[6]For example, systems that exhibit small variations in the frequencies among the cores due to fabrication process variation would probably benefit less from sophisticated asymmetry-aware scheduling algorithms than explicitly asymmetric systems.

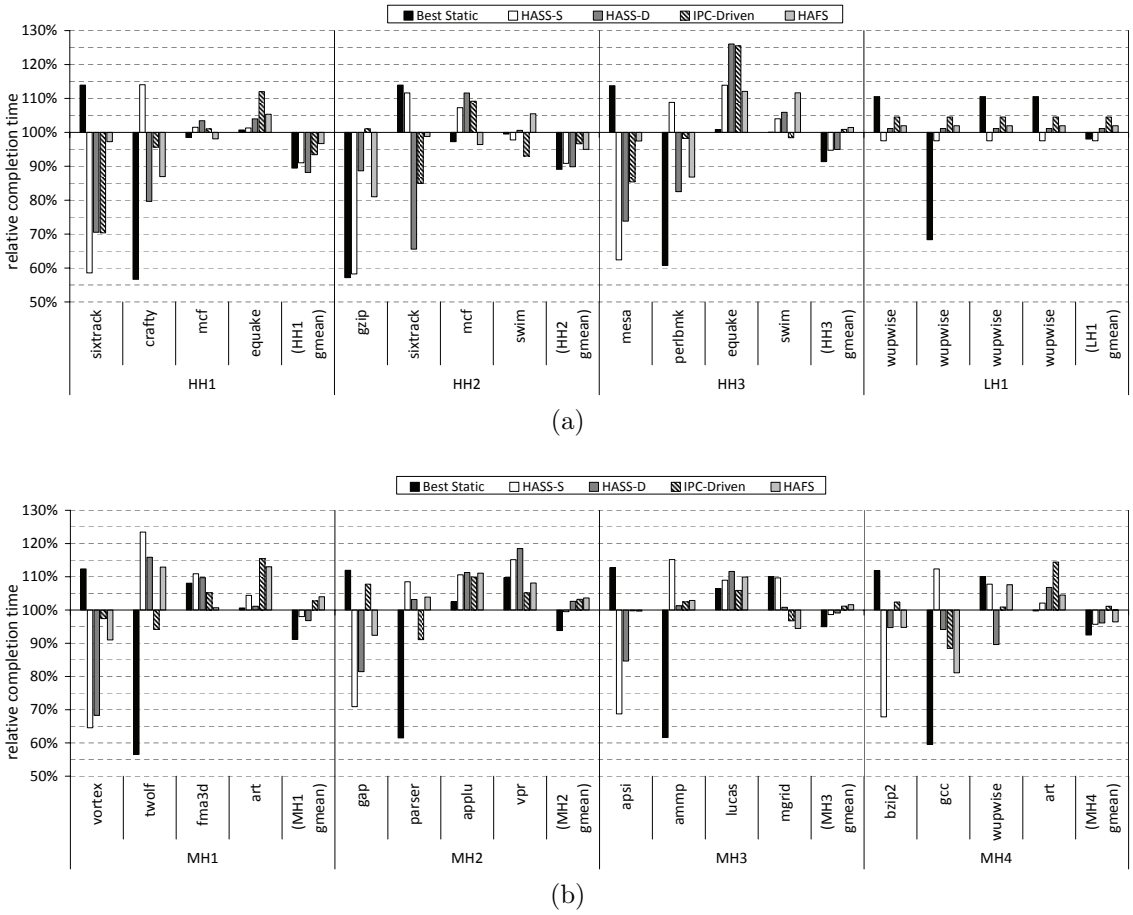


Figure 4.4: Completion times relative to the default metric for workload set #1 on the 4FC-12SC configuration, base workload multiplied by 4 (16 benchmarks in total). (a) HH and LH workloads, (b) MH workloads.

that picking the "wrong" applications did not have a large impact on performance – in many cases it did not matter which applications would be chosen to run on fast cores. As a result, HASS-S performed only 0.5% worse (on average) than the best static assignment. This demonstrates that the HASS-S algorithm is robust even in these conditions difficult for optimization.

Finally, we examine the performance of workload sets #1 and #2 on the 4FC-12SC configuration (Figures 4.4 and 4.5). We expected to see the smallest performance improvements here, because there are proportionally fewer fast cores than on the other systems (only a quarter of the cores is fast as opposed to one half on the other configurations), and also because there is cache sharing.

Examining the results for the HH workloads (Figure 4.4a) we note that HASS-S did not always pick the same application to run on the fast cores as that which was picked by the best static assignment. For example, in the workload {`sixtrack`, `crafty`, `mcf`, `equake`} HASS-S assigned the four copies of `sixtrack` to run on the four fast cores, while the best static assignment picked `crafty`. The differences in sensitivities of `crafty` and `sixtrack` are so small that it is difficult for HASS-S to make this distinction. At the same time, failure to make this distinction does not have a large effect on performance, so HASS-S underperformed the best static

assignment only by 2%.

A similar phenomenon (not picking the same applications to run on the fast cores as those picked by the best static assignment) can be observed for the MH workloads (Figure 4.4b). It is important to note, however, that HASS-S has never made an incorrect choice of running memory-bound applications on the fast cores: it has always correctly picked the CPU-bound applications. The reason for not picking the sames ones as the best static assignment is that the signatures for CPU-bound applications were difficult to distinguish from one another in this moderately heterogeneous workload.

The results for the MH workloads on the 4FC-12SC configuration offer an opportunity to observe the effects of cache sharing, indicating the importance of accounting for this phenomenon in scheduling algorithms. Consider, for instance, the workload {`vortex`, `twolf`, `fma3d`, `art`}. HASS-S chose to run the four copies of `vortex` on fast cores, while according to the best static assignment the four copies of `twolf` should not have been picked. In theory, this "mistake" should not have had much impact on performance, because the sensitivities of `vortex` and `twolf` are very similar. In reality, this "mistake" caused HASS-S to underperform the best static assignment by 7% (although still doing better than default). The reason is cache sharing. `Twolf` is a very cache-sensitive application. That is, its performance suffers when it shares a cache with an aggressive co-runner that generates a lot of cache misses. In this workload, such aggressive applications are `fma3d` and `art`. By running `twolf` on fast cores, as was done under the best static assignment, `twolf` is isolated to run on a separate chip from other benchmarks (recall that the four fast cores in 4FC-12SC are placed on a separate chip), and so it avoids sharing the per-chip last-level cache with the aggressive co-runners. But when `twolf` runs in a slow partition, where the 12 cores are spread across the three chips, it risks sharing a cache with the aggressive `art` or `fma3d`. These results indicate the importance of incorporating the awareness of shared caches into scheduling algorithms for multicore systems.

We now focus our attention on the performance numbers of workload set #2 on the 4FC-12SC configuration (shown in Figure 4.5). Since each workload in this set includes up to sixteen different applications and exhibits a wide diversity in speedup factors (as on the HH workloads), a significant improvement over the default metric can be potentially achieved by asymmetry-aware schedulers. Performance gains delivered by HASS-S are very close to the best static's counterparts for workloads W1, W5, W7 and W8. In fact, execution traces obtained with Dtrace revealed that those gains stemmed from the fact that HASS-S actually chooses the mapping of threads to cores that corresponds to the best static assignment. In the remaining workloads, HASS-S picked a different set of applications to run on fast cores to best static's counterparts, but because the divergences between the speedup factors of the benchmarks mapped to fast cores in both cases are small, HASS-S's "wrong" application mappings do not have a large impact on performance (at most 2.5% for the W3 workload).

We must also highlight that, on the 4FC-12SC configuration, the estimation model

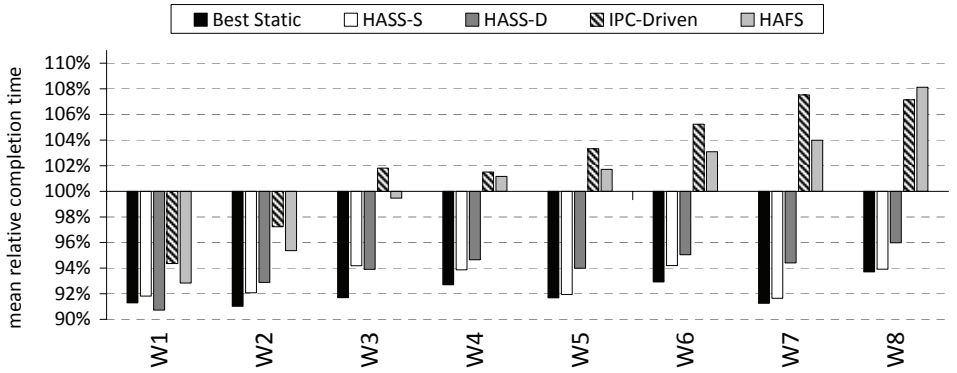


Figure 4.5: Geometric mean of completion times relative to the default metric for workloads in set #2 on the 4FC-12SC configuration.

used by HASS-S is affected by the presence of shared caches. Essentially, the LLC miss rate of the applications may vary due to the sharing of the cache with other threads and, as a result, offline-estimated ratios used by HASS-S may not approximate so accurately the observed ratios during the execution. In other words, the fact that the miss rate may decrease because of cooperative data sharing or increase due to cache contention may lead to overestimation or underestimation of the ratios, respectively. However, previous researchers [62] observed (and so did we) that the *quality* of the miss rate does not change significantly no matter whether the thread shares a cache or runs solo: i.e., if the thread's miss rate is low relative to other threads when it runs solo, its value relative to other threads will stay low when it shares the cache even though it may increase by tens or hundreds of percent relative to its solo value. Similarly, if the thread's miss rate is high it will stay high relative to other threads, regardless of sharing. For that reason, HASS-S is still able to effectively distinguish between memory-intensive and CPU-intensive applications so it correctly classifies the threads even when core and thread counts increase.

Evaluation of HASS-D

Comparing HASS-D to HASS-S, we see that the former performs within 1%, 2.5% and 3.5% range of the latter on the 2FC-2SC-A, 2FC-2SC-B and 4FC-12SC configurations respectively. Traces of the execution of the workloads, collected by means of Dtrace, led us to concluding that both algorithms perform the same thread-to-core mappings for the vast majority of the execution. The reason behind this behavior is two-fold. First, while most applications involved in the evaluation exhibit several program phases, we have not found any application from the SPEC suite alternating between large CPU-intensive phases and large memory-intensive phases. Since the program-phase-detection engine of HASS-D has been deliberately designed to filter out short-term program phases (in an attempt to reduce the number of thread migrations), this algorithm captures primarily long-term phases. For most applications, this leads HASS-D to detecting one large phase that encompasses nearly the entire execution interspersed with a few shorter phases. Second, both algorithms rely on threads' last-level-cache miss rates to estimate the relative benefit from run-

ning a given thread on fast cores rather than on slow cores, so they obtain similar estimates and perform thread assignments accordingly. In summary, the fact that applications do not exhibit many long-term distinct program phases in conjunction with a common model for performance estimates used by both schedulers makes them perform similarly. Furthermore, it is worth highlighting that both algorithms are exposed to similar mispredictions and, when present, they fail to figure out the optimal assignments (see MH1 and MH3 workloads in Figure 4.3b). However, those minor mispredictions do not affect the overall performance significantly.

As opposed to HASS-S, HASS-D is phase aware. Supposedly, being aware of program phases would enable HASS-D to enforce better thread-to-core mappings throughout the execution. Although adjusting thread-to-core assignments dynamically may improve the system-wide efficiency on AMP systems, it may also introduce performance degradation due to additional thread migrations. The negative impact on performance due to these additional migrations may be especially pronounced for highly memory-intensive applications (such as `mcf`,`equake` and `swim`), whose performance suffers significantly when their cache state needs rebuilding after migrations (at least in the private levels of the cache hierarchy). In particular, HASS-S usually outperforms HASS-D when highly memory-intensive programs are included in the workload, such as for workloads W4 to W8 in set #2 (Figure 4.5). Note, however, that not only does migration overhead affect HASS-D, but it has also a negative impact in the performance of any scheduler triggering a non-negligible number migrations, such as IPC-Driven and HAFS. For those schedulers, overhead of as much as 7% over the default metric is introduced for the aforementioned workloads.

Evaluation of the IPC-Driven algorithm

We now turn our attention to the IPC-Driven algorithm. The results for the three hardware configurations and the different workloads are shown in Figures 4.2a-4.4b. The overall (unexpected) conclusion is that the IPC-Driven algorithm performs worse than HASS and the best static assignment. We expected the IPC-Driven algorithm to work better, since unlike the other approaches relies on a real measured speedup factor rather than estimating it. Despite the careful tuning of configurable parameters in the algorithm (the results are shown for the best combination of parameter values), we could not make the IPC-Driven algorithm match the performance of HASS and of the best static assignment. We discovered that the unexpectedly low performance of the IPC-Driven algorithm is due to two problems: (1) inaccurate estimation of the relative benefit that threads derive from running on different core types, and (2) overhead due to migrations performed as part of dynamic performance monitoring.

We illustrate the first problem by analyzing the performance of the HH workload {`sixtrack`, `crafty`, `mcf`, `equake`} on the 2FC-2SC-A configuration. For that workload, the IPC-Driven algorithm achieves the performance improvement of only 6% over default – recall that HASS-S has achieved a 12% performance improvement for this workload!. To understand the root cause of the problem we analyzed how the IPC-Driven algorithm performed thread assignments relative to HASS-S.

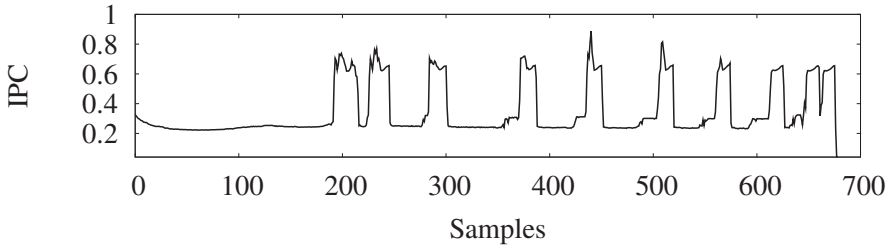


Figure 4.6: IPC over time for `mcf` on the Intel system

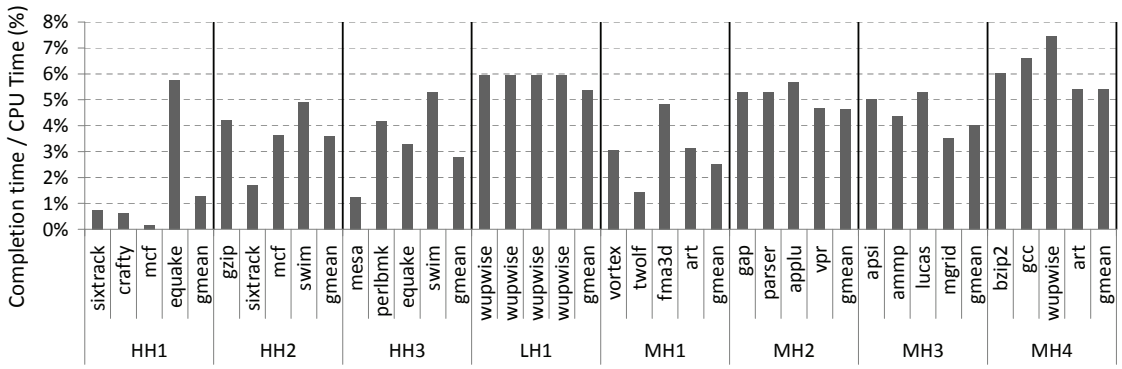


Figure 4.7: The amount by which the wall clock completion time with the IPC-Driven algorithm exceeds the CPU time (user+system) for the 2FC-2SC-A configuration.

Both HASS-S and the best static assignment mapped the two frequency-sensitive applications `sixtrack` and `crafty` to the fast cores, and the two memory-bound applications `mcf` and `equake` to the slow cores. The IPC-Driven algorithm, on the other hand, mapped `mcf` to the fast core roughly 51% of the time, pushing `crafty` to run on the slow core in the meantime (these data were obtained with Dtrace). Although `mcf` does have some high-IPC phases when it makes sense to map it to the fast core (see Figure 4.6), those phases last only 25% of `mcf`'s execution time, not 51%. So 26% of the time `mcf` is not being mapped to the "right" core, which degrades the performance.

The reason for this suboptimal mapping has to do with the unstable nature of phase changes. When `mcf` runs on a fast core during a high-IPC phase and a phase change is detected, it is migrated to a slow core to refresh its IPC ratio. However, as it runs on the slow core, the phase change (and the decrease in the IPC) continues, and so the IPC degradation reflects not only the lower clock frequency of the slow core but the fact that the program has entered an even more memory-bound phase. Ideally, we want the IPC ratio to be computed from the IPCs measured during the *same* program phase. But since each IPC measurement takes a while to perform (the program must run on each core at least several milliseconds in order to amortize for cold cache effects), it is impossible to guarantee that the program will not change a phase during the measurement. As a result, the estimated IPC ratio is inaccurate.

In this particular example we observed that the IPC-Driven algorithm estimated much higher IPC ratios than could be obtained on this hardware. Specifically, `mcf`'s IPC ratio between fast and slow cores was computed to be as high as 2.2 and 2.5

on some occasions. On this configuration, however, the highest possible IPC ratio can be 2.0, because the difference between the frequencies of fast and slow cores is a factor of two. Since the (incorrectly estimated) ratio is too high, the algorithm erroneously decides that `mcf` derives a far more significant benefit from running on the fast core than in reality. As a result, `mcf` is assigned to a fast partition, when in fact it would be more optimal to assign it to a slow partition.

It is very difficult to ensure that the IPCs used to compute the ratio belong to the same phase. Phase changes are difficult to predict at runtime. The problem gets worse if the number of core types, and hence the number of IPC measurements that must be done, is large (recall that the ratio has to be computed for each class of processors). Increasing the `ipc_threshold` did not help, because no single value worked well for all applications.

The reason why this problem did not occur in the original (simulated) evaluation of the IPC-Driven algorithm [2] is that IPC refreshing was not simulated in the same way as it would happen on a real system. IPCs used to compute ratios were obtained from offline IPC traces, and so in contrast with real systems IPCs always corresponded to the same program phase. In other words, in the earlier simulation-based evaluation it was assumed that the IPCs on different core types can be obtained *instantaneously*, while in reality this could not be accomplished.

Another reason why the IPC-Driven algorithm performed worse than expected is the overhead associated with forced thread migrations, which were performed as part of online monitoring. Recall that the IPC-Driven algorithm must periodically refresh the IPC of all threads on all core types. In order to do so, the scheduler forcefully migrates each thread to the cores of different types for IPC measurement. Unfortunately, this creates load imbalance in the system, because as a result of these migrations some cores may have more threads wanting to run on them than others. As a result of a load imbalance, threads running on "overloaded" cores experience longer *CPU wait times* that threads on "underloaded" cores. That is, they spend more time waiting in the CPU runqueue until the core to which they are assigned becomes available. This causes their performance to degrade.

To illustrate this phenomenon we have measured to what extent longer CPU wait times affect the performance under the IPC-Driven algorithm. The CPU wait time is the difference between the wall clock completion time and the total CPU time (computed as the sum of user and system CPU times). So if the CPU wait times were negligible (as it should be on our configuration where the number of threads never exceeds the number of cores), the wall clock time would be roughly equal to the CPU time. Figure 4.7 shows the amount (in percent) by which the wall clock time exceeds the CPU time for the 2FC-2SC-A configuration (results for other configurations are omitted, but they are qualitatively similar). The difference in the wall clock time relative to the CPU time is the overhead due to load imbalance. It can be seen that the IPC-Driven algorithm sacrifices a few percentage points of performance due to load imbalance for almost every workload.

Migration overhead was not detected in the original paper on the IPC-Driven algorithm [2], perhaps because runqueue contention was modeled differently than in

a real scheduler. The paper did not provide sufficient detail about this part of the simulation. Increasing the `ipc_threshold` and `swap_inactivity` period alleviates migration overhead, but at the expense of making the algorithm less phase aware.

In summary, the particular monitoring methodology used in the IPC-Driven algorithm (and in another asymmetry-aware algorithm [3]) suffered from several significant problems. The problems stemmed from the fact that the measurements had to be performed on *every* type of core in the system. Addressing these problems would require a fundamental redesign of the IPC-Driven algorithm. Essentially, we have done this in some way by implementing HASS-D. This algorithm is not subjected to these problems since it estimates performance ratios from measurements obtained on a single core, as opposed to multiple cores. As a result, HASS-D outperforms IPC-Driven across the board.

Evaluation of the HAFS algorithm

In evaluating HAFS, we are first of all interested in investigating whether HAFS accomplishes fair sharing of fast cores among applications. To demonstrate HAFS's fairness property we ran an experiment consisting of several instances of the same application (we chose `mgrid` from the SPEC CPU2000 suite) and measured the fraction of time that each instance spends running on fast cores. (Running four identical applications that have identical completion times simplified data analysis.) We varied the number of concurrent instances from three to ten. The experiments were run on the 2FC-2SC-A configuration. Figure 4.8 shows the results. It can be observed that the fast-core CPU clock cycles are shared equally among the concurrently running instances. When the number of concurrent instances (or threads) equals or exceeds the number of cores, each thread spends 50% of its time running on a fast core. When the number of concurrent instances is three, each thread spends roughly 66% of its time on a fast core, because there are fewer threads competing for fast cores and each one is entitled to its own share.

These results demonstrate two nice properties of the algorithm: first, it ensures fairness in sharing asymmetric CPU resources. Second, it ensures stable and predictable completion times. Recall that with the native scheduler, completion times were highly variable.

The second question we were interested in investigating has to do with performance overhead due to thread migrations performed by HAFS. Thread migrations are an integral part of HAFS or any implementation of the asymmetry-aware round-robin algorithm. Threads *must* be migrated between cores of different types to accomplish fair sharing of fast cores at fine-granular intervals. Although the migration mechanism in HAFS does not cause load imbalance, migrations may still degrade the performance due to disturbing threads' cache affinity [69]. When a thread is moved from one core to another it loses the cache state accumulated in the old core's private caches, and in the old shared cache if the new core does not share a cache with the old one. These overheads have not been investigated in earlier work and we use HAFS as a tool to study them.

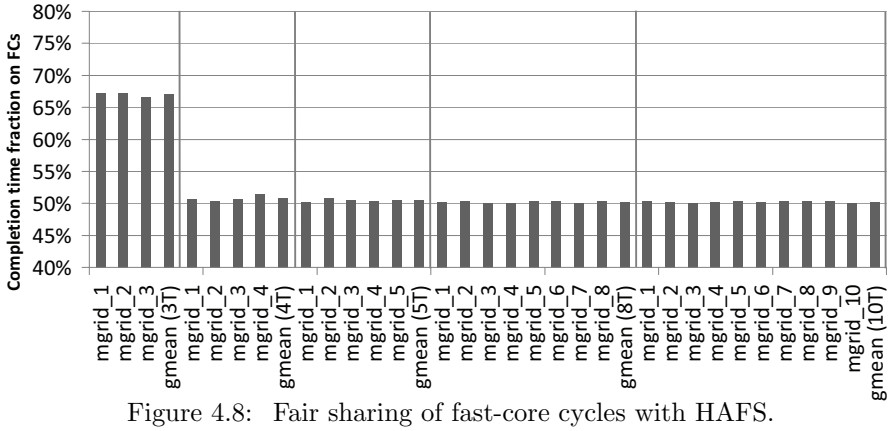


Figure 4.8: Fair sharing of fast-core cycles with HAFS.

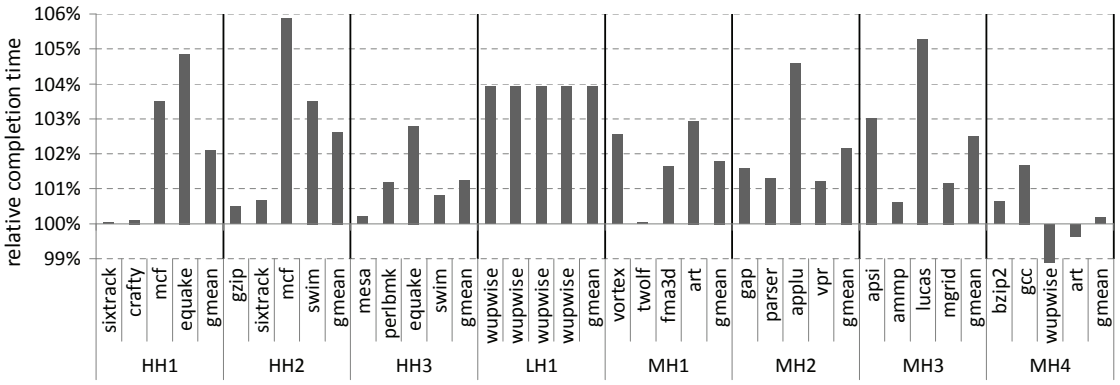


Figure 4.9: Completion times under HAFS normalized to ideal-RR on the 2FC-2SC-A configuration.

To that end, we compare the performance of HAFS with an *ideal-round-robin* (ideal-RR) metric. The ideal-RR metric is computed by combining the previously measured completion times on all core types such that the total time spent on each core type is proportional to how many of those cores are present in the system. Essentially, the ideal-RR metric estimates the completion time for a benchmark in conditions where the cores of different types are shared equally, but when there are no overheads due to migrations. Comparing completion times estimated with the ideal-RR metric to completion times obtained under HAFS enables us to evaluate the migration overheads in HAFS.

Figure 4.9 shows HAFS completion times normalized to the ideal-RR completion times on the 2FC-2SC-A configuration (the results for the other configurations are omitted, but we describe them in the text). The increase in HAFS completion times relative to ideal-RR is the migration overhead. We see that the migration overhead is significant, but not prohibitively large. On this configuration the overhead reached at most 6% for some memory-bound applications. On the 4FC-12SC configuration, where competition for cache was more severe, the overhead reached 25% for one memory-bound application (`twolf`), but hovered around 5-10% for the rest of the applications.

All in all, the migrations required to deliver fair sharing of fast and slow cores do

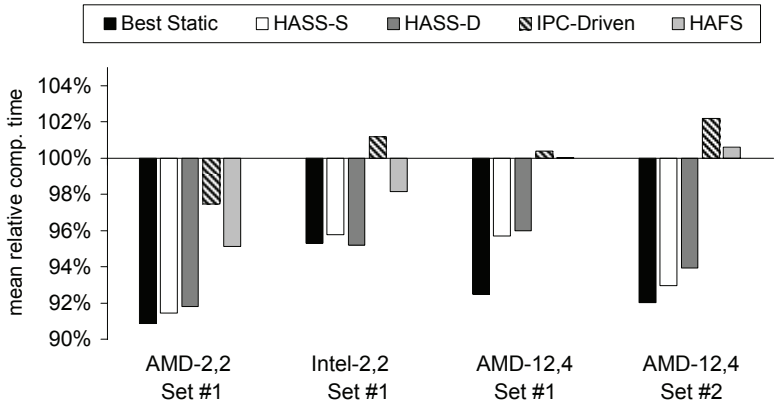


Figure 4.10: Overall reduction in completion time relative to the default metric across configurations and workload sets.

cause overhead. But despite this overhead, HAFS outperforms default on 2FC-2SC-A and 2FC-2SC-B (by 5% and 2% on average respectively) and breaks even with default on the 4FC-12SC configuration (on average for workload set #1, see Figures 4.4a-4.4b). We found that workloads including highly memory-intensive applications, such as W4-W8 from workload set #2, are the most extreme cases, where migration overhead translates into performance degradation with respect to the default metric.

Overall results

Figure 4.10 shows the overall reduction in completion time over the default metric delivered by all the evaluated schedulers across the different asymmetric configurations and workloads. The results show that the performance of HASS-S and HASS-D is very close to the best static's in all cases except one: workload set #1 on 4FC-12SC. In this specific case, the fact that four instances of each application are used when running workload set #1 on 4FC-12SC (sixteen cores) makes the estimated speedup factors of the applications closer, and, as result, the entire workload set is less heterogeneous than the others. All in all, the overall results reveal that both versions of HASS deliver greater performance gains when the workload exhibits enough heterogeneity, and more importantly, that these benefits can still be obtained when the number of threads and cores increase.

Summary

In summary, the results presented in this section lead us to drawing the following conclusions:

- HASS-S is an effective and robust asymmetry-aware scheduling algorithm that is able to differentiate among benchmarks with different architectural properties and assign CPU-intensive applications to fast cores and memory-intensive applications to slow cores.

- It is more difficult for HASS-S to distinguish among the sensitivities of applications whose signatures are very similar, as would be the case with two CPU-intensive applications. While in this case HASS-S often does not match the best static assignment, the performance impact is small, because the wrongly classified applications have very similar speedup factors.

- Cache sharing has an important impact on performance and so it is crucial to incorporate shared cache awareness in HASS-S or any other scheduling algorithm for multicore systems.

- Cross-core migrations required for performance ratio measurements in IPC-Driven often lead to inaccurate IPC ratios and disrupt load balance of the system. We have also showed that HASS-D, the other phase aware algorithm, is not subjected to these problems since it estimates performance ratios from measurements obtained on a single core, as opposed to multiple cores. As a result, HASS-D outperforms IPC-Driven across the board, and so does HASS-S.

- In most cases, HASS-S delivers slightly better performance gains that its dynamic version, HASS-D. This was an unexpected finding, because HASS-D, as opposed to HASS-S, is phase aware and so it can adjust thread-to-core mappings dynamically as applications in the workload go through different program phases. Unfortunately, the additional number of migrations triggered by HASS-D introduce overheads that may significantly reduce the benefits coming from phase-aware thread assignments. Furthermore, we have observed that migration overhead also has a negative impact on the performance of other schedulers like IPC-Driven and HAFS, which trigger a non-negligible number of migrations as well, and the presence of highly memory-intensive applications further aggravates this issue.

- The performance improvements from asymmetry-aware scheduling are especially pronounced on systems where the difference in CPU speeds among the cores of different types is large.

- Fair sharing of fast cores with HAFS comes at a cost, but in most cases the benefits justify this cost.

4.4. Related work

A large body of work has advocated the potential benefits of asymmetric single-ISA processors over symmetric counterparts [3, 1, 4, 5, 13]. These benefits are concisely summarized in an article by Matt Gillespie [8] of Intel, where he lays out some of the background for why this shift towards asymmetric systems is likely to happen, describes the potential variations on the hardware architectures, and the distinct challenges and opportunities. OS scheduling is one of the main challenges, and this is the focus of our thesis.

An asymmetry-aware scheduler would assign threads to cores so as to deliver *specialization*: threads that benefit most from complex cores would be preferentially assigned to these cores, while other threads would be relegated to low-power cores. As stated in Section 2.1, most research efforts that sought to improve the efficiency of AMP systems have exploited primarily two kinds of core specializations: ILP specialization and TLP specialization. In this section we will focus on describing earlier scheduling proposals based on ILP specialization, which cater to the microarchitectural diversity of the workload. An overview of the techniques aimed to exploit TLP specialization will be covered in the next chapter (see Section 5.3).

Most schedulers for performance-asymmetric systems assume cores of two types: fast and slow. Kumar et al. showed that having only two core types is sufficient to deliver the projected benefits of asymmetric systems [1]. The biggest challenge in designing an asymmetry-aware scheduler exploiting ILP specialization is enabling it to determine online the relative benefit that each thread derives from running on the fast core relative to the slow core.

Two of the most well-known scheduling algorithms that employed ILP specialization have been proposed by Becchi et al. [2] and Kumar et al. [3]. Both of them assume a system with two core types ("fast" and "slow") and rely on continuous performance monitoring to determine optimal thread-to-core assignment. Becchi's IPC-Driven algorithm periodically samples threads' instructions per cycle (IPC) on cores of both types to determine the relative benefit for each thread from running on the faster core. Threads with a high fast-to-slow IPC ratio have a high priority in running on the fast core because they are able to achieve a relatively greater speedup there. Kumar's method uses a similar technique, except that the sampling method is made more robust by using more than one sample per thread. In addition, Kumar proposed an algorithm that tries to determine a globally optimal assignment by sampling performance of thread groups rather than making decisions based on the IPCs of individual threads.

Both of these approaches promise significantly better performance than asymmetry-agnostic policies according to simulation-based evaluations, but they are both difficult to use in practice. Their reliance on sampling on *all* core types means that demand for different core types will be unequal. In particular, the smaller the ratio of fast cores to slow cores, the more demand there will be to run on any given fast core for sampling purposes. This creates a workload imbalance and interferes with threads that are "legitimately" running on faster cores. We found this to be a challenging problem in implementing the IPC-Driven algorithm. Since our algorithm relies on per-thread performance profiles, it avoids performance problems related to sampling on different core types and has a much simpler implementation.

Koufaty et al. concurrently with us devised a similar model that also avoids sampling on different core types [24]. The asymmetric system the authors used for the evaluation was different from the one used in this thesis (see Section 3.2 for more information in our emulation approach). While we opted, like many researchers, to emulate asymmetry by setting the cores of a real system to run at different frequencies, Koufaty et al. were able to configure their system such that some cores had a

smaller retirement width than another cores. They were able to do this thanks to proprietary tools that were available to them at Intel. Koufaty also used hardware performance counters to determine the relative speedup, but unlike our approach, his model did not seek to predict the speedup precisely, but rather find performance metric that had a close correlation with the speedup. On their experimental architecture, Koufaty determined that the rate of off-core requests (i.e., last-level cache accesses and misses) had a high correlation with the inverse of the fast-to-slow relative speedup. This statement can be also drawn from the experiments shown in this chapter, since our model to approximate relative speedups based on the last-level cache miss rate enables the HASS scheduler to achieve reasonably good accuracy, and hence effective thread-to-core mappings.

Teodorescu and Torrellas [70] developed an algorithm for optimal assignment in the context of mildly heterogeneous platforms where core differences are caused by within-die process variation. Although performance profiling is still required, a lot of overhead is avoided by assuming that a thread's IPC is the same on all core types. The approach works well when cores are very similar to each other, but unlike our approach, it is generally inapplicable to highly heterogeneous systems.

The schedulers proposed in this chapter leverage knowledge on relative speedups to maximize *system-wide* performance. However, in the event that some processes have a higher priority than others or in scenarios where the system needs to deliver QoS guarantees, the relative speedup could be used as a *complementary* metric to provide better service for prioritized applications with a minimal effect on performance. For example, the scheduler might decide to run low-priority CPU-intensive threads on fast cores rather than high-priority memory-intensive ones, simply because "wasting" fast cores on running memory-intensive instruction streams may lead to significant degradation of overall system performance. Therefore, the relative speedup could be also used to make a trade-off between QoS and system-wide performance.

As we showed in Section 3.2, practical reasons led us to restricting our evaluation to asymmetric systems in which cores just differ in performance due to different clock speeds. It is worth highlighting that algorithms exploiting DVFS, (such as [71], which is the closest to ours) and for DVFS-based asymmetric single-ISA systems, such as the ones proposed in this chapter, address similar problems from different angles. While the former group asks the question: "Which frequency level delivers the best performance/energy trade-off for a given application?", the latter group asks: "Given a set cores with different fixed frequencies, which core turns out to be more efficient to map a specific application?". Therefore, the key difference between our algorithms and DVFS algorithms is that they rely on being able to *adjust frequency of individual cores* and observe the performance of an application under different frequency settings in order to achieve their goals. In our setting, this would be equivalent to running each thread on each core type which requires cross-core migrations. These migrations may introduce performance degradation, so the approach is not suitable for our setting (see Section 4.3.4). For this reason, existing DVFS algorithms do not address the problem that we are solving. Hence, the main takeaway is that asymmetric single-ISA systems need an algorithm that works

without having to run each thread on each core type. Other DVFS-based algorithms, such as [72], which do not require running applications at different DVFS settings, assume that performance of the application scales with CPU frequency (which we showed not to be the case) and so they do not tackle the same problem that we do either.

4.5. Conclusions

In this chapter we presented HASS, a new scheduling algorithm for asymmetric single-ISA multicore systems. The novelty of HASS is in relying on architectural signatures for estimating relative benefits that threads derive from running on different core types. We presented two versions of HASS, HASS-S and HASS-D, which rely on statically and dynamically generated architectural signatures respectively.

HASS consistently improves the performance over an asymmetry-agnostic scheduler when the workload lends itself to asymmetry-related optimizations. It is robust even in the conditions where performance improvements are difficult to obtain. Benefits from asymmetry-aware scheduling algorithms are especially pronounced for workloads where there is a large disparity between applications' architectural properties and on systems with large differences in the speed among different types of cores. When no performance improvements can be expected due to the nature of the workload, HASS never does worse than the asymmetry-agnostic scheduler.

Contrary to our expectations, our implementation of HASS, both the static and the dynamic version, performed better than the IPC-Driven algorithm that relied on actual measured speedup factors as opposed to the estimated ones. We discovered that the IPC-Driven algorithm suffered from inaccuracies and overheads stemming from the need to measure performance on multiple core types. As a result, HASS-S and HASS-D outperformed the IPC-Driven algorithm for every investigated workload. For the sake of providing a more comprehensive experimental evaluation we compared all algorithms to an asymmetry-aware round-robin algorithm HAFS. We found that HAFS outperforms the asymmetry-agnostic default scheduler, but fails to match the performance of more sophisticated asymmetry-aware algorithms for highly heterogeneous workloads.

When comparing both versions of HASS, we found that the usage of offline collected architectural signatures rather than online ones incurs a lot less overhead at runtime, and this leads the static version to delivering greater performance gains. However, in the event signatures are not either available (i.e., not embedded in the application binary) or are not highly representative throughout the execution (e.g., when the application shows large and fairly distinct program phases), online estimated signatures can be effectively used to fill this gap. For that reason, a hybrid version of HASS, which relies on static signatures and resorts to using dynamic ones when they are not either present or representative enough, would deliver performance gains to a wider range of applications.

Overall, we conclude that using architectural signatures rather than direct measurement of performance of each thread on each core time results in less overhead and delivers greater performance gains.

Chapter 5

Catering to Diversity in Thread-Level Parallelism

In the previous chapter we studied the capability of asymmetric CPU designs to improve performance per watt by exploiting the microarchitectural diversity of the workload, where threads that achieve good performance per watt on fast cores get mapped to fast cores, while threads that accomplish poor performance per watt on these cores get assigned to slow cores. In this chapter, by contrast, we focus on how asymmetric designs enable us to tackle a key problem facing the hardware and software communities today: the difficulty in scaling applications on multicore CPUs. AMP systems can mitigate scalability bottlenecks in parallel applications by accelerating serial phases of execution on fast cores. This particular utilization of fast cores (a.k.a. *TLP specialization*, as shown in Section 2.1.2) has been widely recognized and documented in literature [4, 5, 23].

Mitigating scalability bottlenecks is a crucial problem because CPU designs are increasingly turning in the direction of *many-core* – using many simple, slow, low-power cores as opposed to a few complex and powerful cores [50]. Only scalable parallel software can perform well on these many-core processors. Sequential applications or parallel applications limited by serial bottlenecks will perform poorly, because they can utilize only a handful of cores at a time, so having many slow cores is counterproductive. These applications perform better on processors built of a few complex and powerful cores, but power consumption and heat dissipation put into question the practicality of such systems [73]. To mitigate this problem, AMPs include a few complex cores alongside a large number of slow, low-power cores. This way, scalable phases of parallel applications can still harness the multitude of low-power cores achieving high performance per watt, while serial phases (as well as entirely sequential applications) can run on fast cores, thus reducing the effective serialization and improving overall performance.

In order to fully realize the potential of AMPs for mitigating scalability bottlenecks, the operating system scheduler must consider the *amount of parallelism* in the application when making scheduling decisions. Let us provide an example demonstrating why such *Parallelism-Aware* policy will be effective. Consider an AMP system with

one fast core and nine slow cores. Suppose that the fast core delivers roughly twice as much instruction throughput as the slow core for any thread in the workload. Suppose further that we run a workload consisting of one single-threaded application and one scalable parallel application with nine threads. Under a scheduling policy that simply shares fast and slow cores among threads in a round-robin fashion[1], each thread in the workload will spend roughly ten percent of the time on the fast core, and ninety percent of the time on a slow core. Assuming (optimistically) that synchronization and other potential overheads are negligible, each thread will speed-up by a factor of 1.1× relative to running for the entire time on the slow core (this works out since each thread runs 10% of the time at the speedup of 2× and 90% of the time at a the speedup of 1×). As a result, under this naïve scheduling policy, both applications –the single-threaded and the parallel one– will experience a factor of 1.1× speedup on the AMP processor relatively to a symmetric processor where all cores are slow. The average speedup for the workload will be 1.1×. (Note that in our computation of speedup for individual applications we care about *application-wide* speedup, not per-thread speedup, and we assume that the speedup for a multithreaded application is close to the *average* speedup of its threads rather than the aggregate speedup.) Now consider an asymmetry-aware scheduler that maps threads to cores in consideration of application-level parallelism. Under such policy, the single-threaded application will be scheduled on the fast core for the entire time, and the threads of the parallel application will be scheduled on slow cores. As a result, the single-threaded application will speed-up by a factor of 2× relative to running on a slow core, while the parallel application will have a speed-up of 1×. Nevertheless, the average speedup for the workload will be 1.5×, or 40% better than under the naïve policy. This example illustrates the potential for improving system-wide efficiency through the parallelism-aware scheduling policy on AMP hardware[2].

The main contribution of this chapter is the design, implementation and evaluation of the parallelism-aware (PA) policy in a real operating system. Although the benefits of such a policy have been articulated before and even evaluated either analytically or via limited user-level prototypes [4, 5, 23], the design of an operating system scheduling algorithm that delivers this policy to real applications and its evaluation have not been addressed. Therefore, previous work has left unanswered several important questions whose answer would provide crucial insight for designers of future AMP systems. In this chapter, we aim to provide an answer for questions such as:

1. Can the PA policy be implemented without requiring the modification of applications ?
2. Can additional performance benefits be obtained if the application runtime environment is modified to inform the OS scheduler about the dynamics of the application's serial phases?

[1]Existing asymmetry-unaware schedulers in operating systems do not share fast and slow cores in a round-robin fashion, but we assume such a policy in this example to simplify the illustration.

[2]Scenarios where maximizing speedup of individual applications rather than the workload as a whole may also be important, but we do not focus on that problem in this thesis.

3. How do we deal with scenarios where there are multiple applications running on the system?

4. How does the PA policy compare to less sophisticated asymmetry-aware policies that simply attempt to keep fast cores busy or share them among threads in a round-robin fashion?

5. What are the overheads associated with the use of the PA policy and how can they be mitigated?

As stated in Section 2.2.2, one of the biggest challenges in implementing the PA scheduling policy is to enable the OS scheduler to distinguish between scalable phases of parallel applications and sequential phases. In some applications unused threads block during the sequential phase, and by monitoring the application's runnable thread count, which is exposed to the OS by most threading libraries, the scheduler can trivially detect a serial phase. In other applications, however, unused threads busy-wait (or *spin*) during short periods of time, and so the OS cannot detect these phases simply by monitoring the runnable thread count. To address these scenarios we designed PA Runtime Extensions (PA-RTX) – an interface and library enhancements enabling the threading library to notify the scheduler when a thread spins rather than doing useful work. We implemented PA-RTX in a popular OpenMP runtime, which required only minimal modifications to support them. These extensions are general enough to be used with other runtime systems and threading libraries.

In summary, the PA scheduler has the following properties:

- It automatically detects serial phases of parallel applications in those cases where unused application threads block during a serial phase. In these cases, our scheduler delivers asymmetry-enabled performance improvements without the need to modify either the applications or the application runtime environment.

- The scheduler is also designed to interact with the application runtime environment in order to accelerate serial phases where unused threads spin on the CPU during these phases. In this case the scheduler delivers performance gains to a wider range of applications than the base implementation. The application library providing the implementation of synchronization primitives has to be minimally modified to support the runtime extensions, but applications themselves need not be modified unless they use their own implementation of synchronization primitives.

- The scheduler supports multi-application workloads. If there are multiple applications running on the system the scheduler shares the fast cores among them in a sensible fashion. For example, if several applications have threads that must be scheduled on fast cores according to the PA policy and the number of such threads exceeds the number of fast cores, the scheduler fairly shares the fast cores among those threads.

- In order to mitigate the overhead of potentially costly migrations, the scheduler is *topology aware*: whenever it must migrate a thread from one core to another, it will attempt to arrange a migration that does not require crossing the boundary of the memory hierarchy domain. As we will show later, this reduces the overhead of migrations.

We implemented the PA algorithm in a real operating system, OpenSolaris, and evaluated it on real multicore hardware where asymmetry was emulated by setting the cores to run at different frequencies. We used a variety of sequential and parallel applications drawn from several benchmark suites to evaluate the scheduler. We compare PA to two other simple asymmetry-aware scheduling algorithms that either share fast and slow cores among threads in a round-robin fashion or ensure that fast cores do not go idle before slow cores. To further underscore the benefits of TLP specialization, we also compare PA with a previously proposed asymmetry-aware algorithm exploiting ILP specialization (see Section 2.1.1 for more information on this specialization technique).

As a result of our evaluation, we found that applications with large serial bottlenecks (≈40-60% when measured as the fraction of the execution spent in serial phases) reap significant performance improvements from the PA scheduler. These benefits are obtained without modifications to applications or to the runtime environment. If the application runtime is minimally extended to use the PA runtime extensions, some additional improvements can also be accomplished even when serial bottlenecks are hidden by *spinning*. We also observed that for workloads consisting of only a single application, benefits similar to those delivered by the PA algorithm can be obtained from less sophisticated asymmetry-aware algorithms, but for multi-application workloads these simple algorithms fail to deliver performance improvements comparable to PA. In the latter scenario, PA delivers up to 40% better performance over simple asymmetry-aware algorithms. This demonstrates that an algorithm like PA is essential for efficient utilization of AMP hardware in realistic scenarios. Furthermore, we found that previously proposed asymmetry-aware algorithms, which we used for comparison, also do well in some cases, but unlike our parallelism-aware algorithms they do not perform well across the board, because they fail to consider the parallelism of the application.

In evaluating the overhead, we observed that the PA algorithm, as well as other asymmetry-aware algorithms, introduces some additional latency due to the need to migrate threads between fast and slow cores in order to accomplish its goals. Such migrations are fundamental to asymmetry-aware algorithms in general and cannot be eliminated entirely. Upon evaluating these overheads we found that they can be significant (up to 18%) if the fast core is placed in a different memory hierarchy domain from slow cores, but a hardware configuration where a fast core shares a memory hierarchy domain with several slow cores coupled with a topology-aware scheduler practically eliminates these overheads.

Our overall conclusion is that the benefits of AMP processors related to mitigating serial bottlenecks in applications can be realistically delivered to real software via modest changes to the operating system scheduler, and in many cases without the

need to modify the application runtime environment. Additional performance improvements can be accomplished through interactions with the application runtime and topology-aware system design.

The remainder of the chapter is structured as follows. Section 5.1 presents the design and implementation of the PA scheduling algorithm. Section 5.2 presents experimental results. Section 5.3 discusses related work. Section 5.4 summarizes our findings.

5.1. Design and implementation

In Section 5.1.1, we describe two parallelism-aware algorithms proposed in this thesis: PA and MinTLP. In Section 5.1.2, we describe the runtime extensions to PA (PA-RTX). A brief description of other asymmetry-aware algorithms that we use for comparison is provided in Section 5.1.3.

5.1.1. PA and MinTLP algorithms

Our algorithms assume an AMP system with two core types: *fast* and *slow*. Previous studies concluded that supporting only two core types is optimal for achieving most of the potential gains on AMPs [1], so we expect this configuration to be typical of future systems. More core types may be present in future systems due to variations in the fabrication process. In that case, scheduling must be complemented with other algorithms, designed specifically to address this problem [70].

The PA and MinTLP algorithms rely on the design approach to asymmetry-aware schedulers described in Section 2.3.2, which is based on the core partition abstraction. Therefore, fast and slow cores are separated into fast and slow partitions respectively, and threads are assigned to partitions (sets of cores where the thread is allowed to run) rather than to individual cores. For the sake of simplicity, the description of the algorithms provided in this section assumes that only two core partitions are used: FAST and SLOW.

Both PA and MinTLP monitor applications' runnable thread counts and react to changes in these by potentially triggering thread migrations between partitions. For example, if an application enters a sequential phase and the only runnable thread is mapped to a slow core by then, PA will migrate this thread onto a fast core in an attempt to effectively accelerate the sequential phase. Although our evaluation focuses on multi-threaded single-process applications, the PA and MinTLP algorithms can be seamlessly extended to support multi-process software, such as MPI applications, by using high-level abstractions provided by modern operating systems, such as *process groups*[3].

[3]In POSIX-conformant operating systems, a *process group* denotes a collection of one or more processes. Process groups are used to control the distribution of signals.

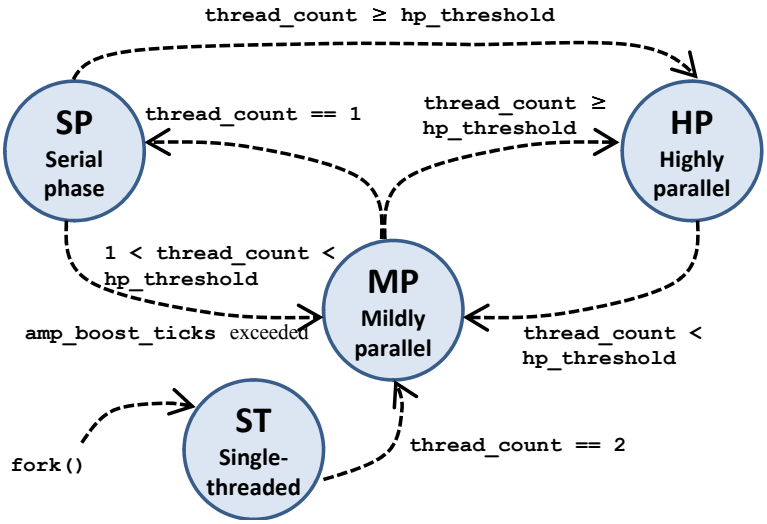


Figure 5.1: Application Classes.

The goal of the algorithms is to decide which threads should run on fast cores and which on slow cores. In MinTLP, this decision is straightforward: the algorithm selects applications with the smallest thread-level parallelism (hence the name MinTLP) and maps threads of these applications to fast cores. Thread-level parallelism is determined by examining the number of runnable (i.e., not blocked) threads. If not enough fast cores are available to accommodate all these threads, some will be left running on slow cores. MinTLP makes no effort to fairly share fast cores among all "eligible" threads. This algorithm is very simple, but not always fair.

The other proposed algorithm, PA, is more sophisticated. Applications and their respective threads are assigned to classes that determine the partition where the thread will run. The class transition diagram is shown in Figure 5.1. The PA scheduler determines the application class based on the number of runnable threads in the application. The scheduler detects the changes in the number of runnable threads in the application and transitions each thread in the application into the appropriate class. A newly created application, after forking its first thread, is assigned by the scheduler to the class ST. This class represents single-threaded applications. If that application increases its thread count to two, the scheduler assigns it to the MP class. This class is for mildly parallel applications. If an application increases its thread count to `hp_threshold` (a configurable parameter whose setting is discussed later on), the scheduler assigns it to the class HP – a class for highly parallel applications. An HP application whose thread count falls below `hp_threshold` is assigned back into the MP class. An MP application whose thread count reaches one is assigned into the SP class – a class designated to represent serial phases of parallel applications. An SP thread is assigned back to the MP class after running in the FAST partition for `amp_boost_ticks`. The reason for this transition will be explained later.

We now motivate our classification scheme. At the base of our classification are two classes: ST – for threads running sequential applications, and HP – for threads of highly parallel applications. These threads must be distinguished from each other on an AMP system. Given that, without special treatment, threads of mildly parallel applications would fall into the HP class, we introduced a separate class, MP, for applications with only a handful of threads. We opted to do so because MP applications do not benefit from running on a large number of slow cores as much as highly parallel applications. Therefore, we wanted to give mildly parallel applications a high priority for running on fast cores. We also have a separate class SP for the thread of an application that has just entered a serial phase, because we wanted to maximally mitigate the performance effects of scalability bottlenecks. As a result, we arrived at the classification scheme presented in Figure 5.1.

Initial mapping of newly created threads is performed such that the FAST partition is populated before the SLOW partition and the balance of load across partitions is preserved. Subsequent *migrations* will enforce the Three Rules of the PA algorithm: (1) ST and MP threads must have a higher priority for running on fast cores than HP threads, (2) SP threads must be moved to the FAST partition as quickly as possible without monopolizing it for long periods of time, (3) load balance and fair distribution of CPU cycles among threads must be preserved.

Rule 1 may be broken either when a thread transitions between a runnable and a non-runnable state (the thread blocks or becomes active) or in the event that a thread transitions into a different parallelism class. In both cases PA may need to perform cross-partition migrations. To this end, the scheduler follows a carefully crafted mechanism devised to avoid load imbalance. In the former scenario, PA guarantees that Rule 1 holds true by migrating one selected thread between the FAST and SLOW partitions. Although a change in the number of runnable threads assigned to FAST and SLOW partitions does not always result in breaking Rule 1, PA may still need to migrate a thread to the opposite partition to ensure load balance across partitions (see Section 2.3.2 for more information on asymmetry-aware load balancing). In the latter scenario, the scheduler enforces the rule by swapping two threads between the FAST and SLOW partition.

A swap works as follows. A thread that must be migrated to another partition is inserted by the scheduler into a list of *swap candidates*. There are such lists for FAST-to-SLOW candidates and for SLOW-to-FAST candidates. A swap is performed between the top candidates in the two lists. If the scheduler finds no matching swap candidate in the opposite list it keeps this thread running in its old partition until a matching swap candidate is found. List insertion is performed as soon as the thread is ready for migration, but swaps are performed periodically.

To enforce the Three Rules of the PA algorithm, thread swaps between the FAST and SLOW partitions are required in the following four scenarios.

Scenario 1. In this scenario, the ratio of ST and MP threads to fast cores is greater than the ratio of HP threads to slow cores. This means that it would be impossible to assign all these threads to the FAST partition without unbalancing the load. To avoid this, the PA algorithm runs ST and MP threads in both the

Table 5.1: Priority of swap candidates

FAST-to-SLOW candidates	SLOW-to-FAST candidates
HP	SP
ST, MP expired	ST, MP expired
ST, MP unexpired	ST, MP unexpired

FAST and SLOW partitions, but ensures that all these threads share fast cores equally. This is accomplished as follows. The scheduler associates with each thread an "expiration" counter to track how many cycles it has spent running in the FAST partition. When that counter reaches a certain threshold, the scheduler marks the thread as "expired" and inserts it into the FAST-to-SLOW swap list. When a matching candidate appears in the SLOW-to-FAST list the thread will be swapped. Similarly, when an ST or MP thread has accumulated a threshold number of cycles in the SLOW partition the scheduler marks it as "expired" and inserts it into the SLOW-to-FAST swap list. Note that this is a similar technique to the one used by the HAFS scheduler, presented in Chapter 4. As we demostrated in Section 4.3.4, this mechanism ensures that all threads receive an equal share of cycles on fast cores, while preserving the load balance.

Scenario 2. An HP application decreases its thread count below *hp_threshold* and is assigned to the MP class. All threads of that application that were running in the SLOW partition will be inserted into the SLOW-to-FAST swap list. The scheduler will also turn on the "expiration" counters for these threads.

Scenario 3. An MP application whose thread count decreases to one is assigned to the SP class. If that thread is running on a slow core, it will be inserted into the SLOW-to-FAST candidate list for migration.

Scenario 4. An MP application running in the FAST partition increases its thread count and is assigned to the HP class. In that case, the threads of this application must be migrated to the SLOW partition to respect Rule 1 of the algorithm. To that end, the scheduler will insert the threads into the FAST-to-SLOW swap list.

To respect the rules of the PA algorithm swap candidates are *prioritized* according to the rules summarized in Table 5.1. An HP thread running in the FAST partition must be swapped as soon as possible to the SLOW partition if there are MP, ST or SP threads wanting to migrate to the FAST partition, so it will have the top priority for a FAST-to-SLOW swap (see row 1, column 1 in Table 5.1). An SP thread running in the SLOW partition must be migrated immediately to the FAST partition, so SP threads will have the top priority for a SLOW-to-FAST swap (row 1, column 2). To avoid the monopolization of the FAST partition by SP threads, such threads are downgraded to the MP status after they have run on a fast core for `amp_boost_ticks` clock ticks (before the expiration of `amp_boost_ticks` an SP thread will never be migrated from the FAST partition – this feature ensures the acceleration of serial phases). ST and MP threads that have reached the expired status are at the next priority level for swapping, and ST and MP threads that have not reached the expired status are at the lowest level.

We now show that the PA algorithm respects its Three Rules. Rule 1 will be respected since if there is an HP thread in the FAST partition and an ST or MP thread in the SLOW partition, they will always be swapped according to the priority order in Table 5.1. Rule 2 will be respected, because an SP thread running in the SLOW partition will always be swapped with *any* non-SP thread in the FAST partition. Furthermore, monopolization of the FAST partition by SP threads will not be allowed, since they will be downgraded to the MP class after `amp_boost_ticks` clock ticks. Rule 3 will be respected, because the assignment of newly created threads ensures load balance, and subsequent thread migrations are performed via swaps, which never make the load balance worse.

Finally, it is worth highlighting that the performance of the PA scheduler is sensitive to the settings of the aforementioned *tunnables* (`amp_boost_ticks` and `hp_threshold`), so we have carried out an exhaustive evaluation of the parameter space and picked the ones that delivered the best overall performance. We set `amp_boost_ticks` to one hundred time slices (1 second) and `hp_threshold` was set to one greater than the number of fast cores. Although we followed an empirical approach to selecting an appropriate value for `hp_threshold` here, it is also possible to select this value by estimating the minimum number of threads for which an application obtains little or no benefit from mapping as many of its threads as possible to fast cores. We will elaborate on this technique in Chapter 6.

5.1.2. PA runtime extensions

The base PA algorithm introduced so far relies on monitoring runnable thread count to detect transitions between serial and parallel phases in the application. However, conventional synchronization primitives found in most threading libraries use an adaptive two-phase approach where unused threads busy wait for a while before blocking to reduce context-switching overheads. While blocking is coordinated with the OS, making it possible to detect phase transitions, spinning is not. Reducing the spinning phase enables the OS to detect more serial phases. However, in our context it may also lead to excessive migrations and cause substantial overheads (as soon as a fast core becomes idle PA and MinTLP will immediately migrate a thread to this core). In the event these busy-waiting phases are frequent, it is helpful to give the scheduler some hints that would help it to avoid mapping spinning threads to fast cores. To that end, we propose two optimizations, which can be implemented in the threading library (applications themselves need not be changed).

Spin-then-notify mode

Our first proposal is a new *spin-then-notify* waiting mode for synchronization primitives. Its primary goal is to avoid running spinning threads on fast cores and save these "power-hungry" cores for other threads. In this mode, the synchronization primitive notifies the operating system via a system call after a certain *spin threshold* that the thread is busy-waiting rather than doing useful work. Upon notification,

the PA scheduler marks this thread as a *candidate* for migration to slow cores. We have opted to mark threads as migration candidates instead of forcing an immediate migration since this approach avoids premature migrations and allows a seamless integration with the PA and MinTLP swapping mechanisms. The synchronization primitive also notifies the scheduler when a spinning thread finishes the busy wait. In Section 5.2.2, we explore the advantages of using the new spin-then-notify mode. For this purpose we have modified the OpenMP runtime system to include this new mode in the basic waiting function used by high-level primitives such as mutexes or barriers.

Another potentially useful feature of this primitive may arise in the context of scheduling algorithms that map threads on AMP systems based on their relative speedup on fast vs. slow cores (as those covered in Chapter 4). These algorithms typically measure performance of each thread on fast and slow cores and compute its performance ratio, which determines the relative speedup [2, 3]. If a thread performs busy-waiting it can achieve a very high performance ratio, since a spin loop uses the CPU pipeline very efficiently[4]. As a result, the proposed algorithms would map spinning threads to fast cores despite they are not doing useful work. Even though these implementation issues could be solved via additional hardware support [75], a spin-then-notify primitive could help avoid the problem without needing extra hardware.

Exposing the master thread

We have also investigated a simple but effective optimization allowing the application to communicate to the kernel that a particular thread must have a higher priority in running on a fast core. This optimization was inspired by the typical structure of OpenMP *do-all* applications. In these applications, there is usually a *master thread* that is in charge of the explicit serial phases at the beginning, in between parallel loops, and at the end of the application (apart from being in charge of its share of the parallel loops). Identifying this master thread to the kernel enables the scheduler to give it a higher priority on the fast core simply because this thread will likely act as the "serial" thread. This hint can speed up do-all applications even without properly detecting serial phases. Our PA Runtime Extensions enable the runtime system to identify the master thread to the scheduler via a new system call. If the pattern of the application changes and another thread gets this responsibility, the same system call can be used to update this information.

To evaluate this feature, we have modified the OpenMP runtime system to automatically identify the thread executing the *main* function as the master thread to the kernel, right after initializing the runtime environment. In the same way as the implementation of spin-notify mode, only the OpenMP library needs to be modified, not requiring any change in the applications themselves. Upon receiving this notification, the PA scheduler tries to ensure that the master thread runs on a

[4]Best practices in implementing spinlocks dictate using algorithms where a thread spins on a local variable [74], which leads to a high instruction throughput.

fast core whenever it is active, but without permanently binding the thread to that core as would be done with other explicit mechanisms based on thread affinities. This way, PA still allows different threads to compete for fast cores according to its policies.

5.1.3. The other schedulers

We compare PA and MinTLP to three other schedulers proposed in previous work. The asymmetry-aware Round-Robin (RR) algorithm fair-shares fast cores among all threads[5]. BusyFCs is a simple asymmetry-aware scheduler that guarantees that fast cores never go idle before slow cores [14]. Static-IPC-Driven, which we describe in detail below, assigns fast cores to those threads that experience the greatest relative speedup (in terms of instructions per second) relative to running on slow cores [2]. We implemented all these algorithms in OpenSolaris. Our baseline for comparison is the asymmetry-agnostic default scheduler in OpenSolaris, referred to hereafter as Default.

The Static-IPC-Driven scheduler is based on the design proposed by Becchi and Crowley [2]. This algorithm was described in detail in Section 4.2.3. Overall, thread-to-core assignments in that algorithm are done based on per-thread IPC ratios (quotients of instruction-per-cycle counts on fast and slow cores), which determine the relative benefit of running a thread on a particular core type. Threads with the highest IPC ratios are scheduled on fast cores while remaining threads are scheduled on slow cores. In the original work [2], the IPC-driven scheduler was simulated. This scheduler samples threads' IPC on cores of all types whenever a new program phase is detected. Recall that, in the previous chapter we reported on our experience evaluating a real-implementation of this algorithm, and showed that such sampling caused large overheads stemming from frequent cross-core thread migrations (as shown in Section 4.3.4). To avoid these overheads, we have implemented a static version of the IPC-driven algorithm, where the IPC ratios of all threads are measured *a priori*. This makes IPC ratios more accurate in some cases and eliminates much of the runtime performance overhead. Therefore, the results of the Static-IPC-Driven scheduler are somewhat optimistic and the speedups of PA and MinTLP relative to Static-IPC-Driven are somewhat pessimistic.

5.2. Experiments

The evaluation of the PA algorithm was performed entirely on the AMD-16 platform, presented in Section 3.3. Such a platform consists of sixteen cores organized into four chips sharing an L3 cache (quad-core "Barcelona" CPUs). On this system, each core is capable of running at a range of frequencies from 1.15 GHz to 2.3

[5]To the best of our knowledge, the RR algorithm was first proposed in [2]. For the evaluation in this chapter, we used our real-world implementation of such a RR policy: the HAFS algorithm, described in Section 4.2.4.

GHz. Since each core is within its own voltage/frequency domain, we are able to vary the frequency for each core independently. We experimented with asymmetric configurations that use two core types: "fast" (a core set to run at 2.3 GHz) and "slow" (a core set to run at 1.15 GHz). We also varied the number of cores in the experimental configurations by disabling some of the cores.

We used three AMP configurations in our experiments: (1) 1FC-12SC – one fast core and twelve slow cores, the fast core is on its own chip and the other cores on that chip are disabled; (2) 4FC-12SC – four fast cores and twelve slow cores, each fast core is on a chip with three slow cores; (3) 1FC-3SC – one fast core, three slow cores, all on one chip. Not all configurations are used in all experiments.

Although thread migrations can be effectively exploited by asymmetry-aware schedulers (e.g., to map sequential parts of parallel applications on fast cores), the overhead that they may introduce can lead to performance degradation. Since we also aim to assess the impact of migrations on performance, we opted to select the default asymmetry-unaware scheduler used in OpenSolaris (we refer to it as Default henceforth) as our baseline scheduler. Despite the fact that Default keeps threads on the same core for most of the execution time and thus minimizes thread migrations, its asymmetry-unawareness leads it to providing much more unstable results from run to run than the ones observed for the other schedulers. For that reason, a high number of samples were collected for this scheduler in an attempt to capture the average behavior more accurately. Overall, we found that Default usually fails to schedule single-threaded applications and sequential phases of parallel application on fast cores, especially when the number of fast cores is much smaller than the number of slow cores, such as on the 1FC-12SC and 4FC-12SC configurations.

We evaluate the base implementation of the PA algorithm as well PA with Runtime Extensions. We compare PA to RR, BusyFCs, Static-IPC-Driven, Min-TLP and to Default. In all experiments, each application was run a minimum of three times, and we measure the average completion time. The observed variance was small in most cases (so it is not reported) and where it was large we repeated the experiments for a larger number of trials until the variance reached a low threshold. In multi-application workloads, the applications are started simultaneously and when an application terminates it is restarted repeatedly until the longest application in the set completes at least three times. We report performance as the speedup over Default. The geometric mean of completion times of all executions for a benchmark under a particular asymmetry-aware scheduler is compared to that under Default. The performance graphs shown in this section report the wall clock speedup (in percentage) for each application. For workloads consisting of two or more applications, the wall clock speedup for the workload as a whole is also provided.

In all experiments, the total number of threads (sum of the number of threads of all applications) was set to match the number of cores in the experimental system, since this is how runtime systems typically configure the number of threads for the CPU-bound workloads that we considered [26].

The remainder of the evaluation section is divided into four parts. In Section 5.2.1, we introduce the applications and workloads used for evaluation. In Section 5.2.2,

Table 5.2: Classification of selected applications.

Categories	Benchmarks
HP-CI	EP(N), vips(P), fma3d(O), ammp(O), RNA(I), scalparc(M), wupwise (O)
HP-MI	art(O), equake(O), applu(O), swim(O)
PS-CI	BLAST(NS), swaptions(P), bodytrack(P), semphy(M), FT(N)
PS-MI	MG(N), TPC-C(NS), FFTW(NS)
ST-CI	gromacs(C), sjeng(C), gamess(C), gobmk(C), h264ref(C), hmmer(C), namd(C)
ST-MI	astar(C), omnetpp(C), soplex(C), milc(C), mcf(C), libquantum(C)

we evaluate PA runtime extensions. In Section 5.2.3, we evaluate multi-application workloads. Finally, in Section 5.2.4 we study the overhead.

5.2.1. Workload selection

We used applications from PARSEC [76], SPEC OMP2001, NAS [77] Parallel Benchmarks and MineBench [78] benchmark suites, as well as the TPC-C benchmark implemented over Oracle Berkeley DB [79], BLAST – a bioinformatics benchmark, FFT-W – a scientific benchmark performing the fast Fourier transform, and RNA – an RNA sequencing application. For multi-application workloads we also used sequential applications from SPEC CPU2006.

We classified applications according to their architectural properties: memory intensive (MI) or compute intensive (CI), as well as according to their parallelism: highly parallel (HP), partially sequential (PS) and single-threaded (ST). Memory-intensity was important for fair comparison with Static-IPC-Driven. As shown in Section 2.1.1, CI applications have a higher relative speedup on fast cores and so it was important to include applications of both types in the experiments. Parallelism class was determined by tracing execution via OpenSolaris' DTrace framework and measuring the fraction of time the application spent running with a single runnable thread. Parallel applications where this fraction was greater than 7% were classified as PS, whereas the rest were classified as HP. The ST class includes sequential applications. Table 5.2 shows the classification of our selected applications according to these classes. The text in parentheses next to the benchmark name indicates the corresponding benchmark suite: O –SPEC OMP2001, P– PARSEC, M – Minebench, N– NAS, C – SPEC CPU2006, and NS – other benchmarks not belonging to any specific suite.

By default, all OpenMP applications were compiled with the native Sun Studio compiler. In order to evaluate PA Runtime Extensions (Section 5.2.2) we had to modify the OpenMP runtime system but the source code for the Sun Studio OpenMP runtime system was not available to us. For that reason, we resorted to using the Linux version of the GCC 4.4 OpenMP runtime system in OpenSolaris[6]. Nevertheless, we observed that the performance of OpenMP applications with Sun Studio and GCC is similar.

[6]Using such a version of the runtime system required augmenting OpenSolaris with a Linux compatible `sys_futex` syscall.

Table 5.3: Multi-application workloads, Set #1.

Workload name	Benchmarks
STCI-PSMI	gamess, FFTW (12,15)
STCI-PSCI	gamess, BLAST (12,15)
STCI-PSCI(2)	hmmer, BLAST (12,15)
STCI-HP	gamess, wupwise (12,15)
STCI-HP(2)	gobmk, EP (12,15)
STMI-PSMI	mcf, FFTW (12,15)
STMI-PSCI	mcf, BLAST (12,15)
STMI-HP	astar, EP (12,15)
PSMB-PSCI	FFTW (6,8), BLAST (7,8)
PSMB-HP	FFTW (6,8), wupwise_m (7,8)
PSCI-HP	BLAST (6,8), wupwise_m (7,8)
PSCI-HP(2)	semphy (6,8), EP (7,8)

Table 5.4: Multi-application workloads, Set #2.

Workload name	Benchmarks
2STCI-2STMI-1HP	gamess, h264ref, astar , soplex, wupwise (12)
4STCI-1HP	gromacs, gamess, namd, gobmk, EP (12)
3STCI-1STMI-1PSCI	gamess, hmmer, gobmk, soplex, semphy (12)
2STCI-1STMI-1PSMI-1HP	gamess, h264ref, soplex, FFTW (6), equake (7)
3STCI-3STMI-1HP	gromacs, sjeng, h264ref, libquantum, milc, omnetpp, EP (10)
3STCI-3STMI-1PSCI	gromacs, sjeng, h264ref, libquantum, milc, omnetpp, BLAST (10)

Both OpenMP and POSIX threaded applications explored in Sections 5.2.3 and 5.2.4 run with adaptive synchronization modes; as such sequential phases are exposed to the operating system in both cases. In these sections, we do not use runtime extensions with parallelism-aware algorithms. All OpenMP applications run with the default adaptive synchronization mode used by GCC 4.4 unless otherwise noted (Sun Studio can be easily configured to use a similar adaptive mode). POSIX threaded applications (such as `BLAST` or `bodytrack`) use full blocking modes on all synchronization primitives but on those related to POSIX standard mutexes and synchronization barriers, where an adaptive implementation is provided by OpenSolaris. Unlike OpenMP applications, threads of POSIX applications spin for shorter periods of time before blocking on those adaptive synchronization primitives (these are the default parameters used in OpenSolaris).

For Section 5.2.2 we selected ten OpenMP applications: `art`, `applu`, `fma3d`, `ammp`, `FT`, `MG`, `scalparc`, `semphy` and `RNA`. These applications were chosen to cover a wide variety of sequential portions. In the overhead section we analyze ten parallel applications across the aforementioned classes: three HPCI (`RNA`, `wupwise` and `vips`), two HPMI (`swim` and `applu`), three PSCI (`swaptions`, `bodytrack` and `BLAST`) and two PSMI applications (`TPC-C` and `FFTW`) .

For Section 5.2.3, we constructed two sets of multi-application workloads. The first set, shown in Table 5.3, comprises twelve representative pairs of benchmarks across the previous categories mentioned above. For the sake of completeness, we experimented with additional multi-application workloads with more than two applications. Table 5.4 shows this second set, consisting of six workloads.

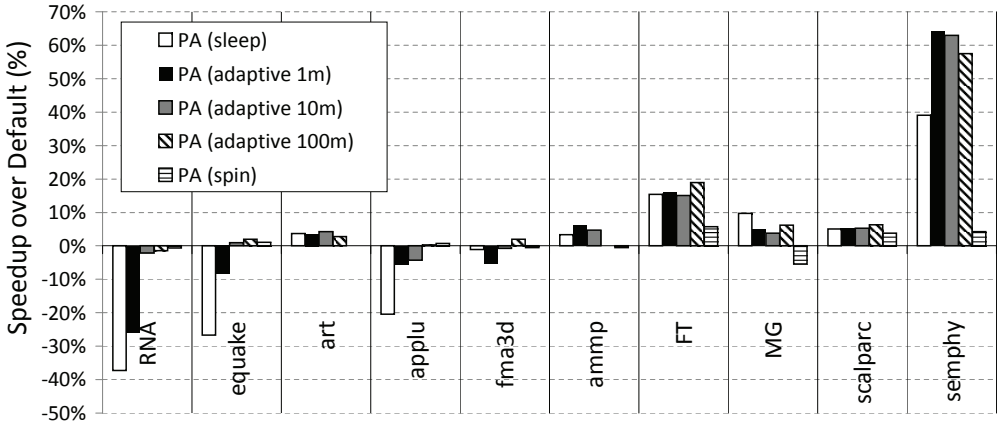


Figure 5.2: Speedup from PA using sleep, spin and adaptive modes with different *blocking thresholds*.

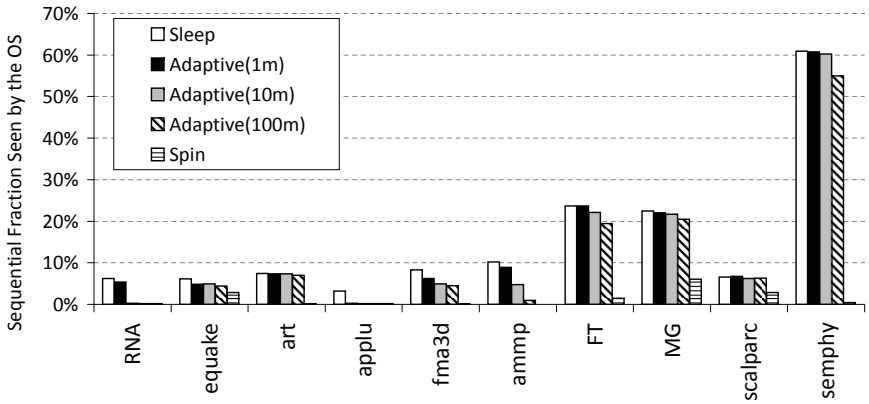


Figure 5.3: Variations in the sequential fraction seen by the OS when varying the synchronization mode and *blocking threshold*.

5.2.2. PA runtime extensions

We begin by investigating the effect on performance when using different synchronization waiting modes under the PA scheduler. In these experiments we demonstrate that using a low blocking threshold effectively exposes sequential phases to the scheduler, but performance can also suffer if the threshold is set too low. Then, we evaluate PA-RTX and show that it offers comparable performance to purely adaptive approaches and in some cases even improves it.

In the following experiment we used the 1FC-12SC configuration and tested three different waiting modes: spin, sleep and adaptive. In spin mode unused threads busy-wait for the entire time; in sleep mode, they block immediately. We studied the effects of various synchronization modes on all asymmetry-aware schedulers, but since our results showed that across the schedulers the effects are largely the same, we present the data for the PA scheduler only.

Figure 5.2 shows the results. PA runtime extensions are *not* used in this case. When

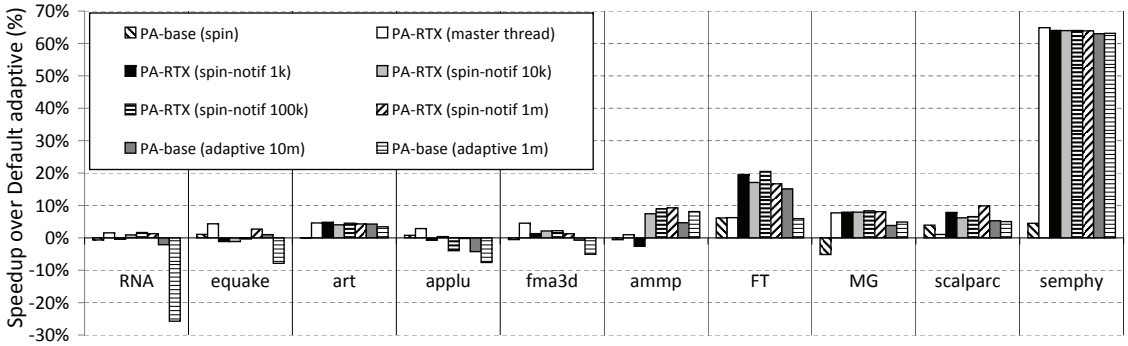


Figure 5.4: Speedup from PA with Runtime Extensions.

the spin mode is used, the base PA algorithm delivers hardly any speedup, because it is not aware of the sequential phases. With the adaptive and sleep modes, applications on the right side of the chart experience noticeable speedup. They have large sequential phases and switching to the adaptive or sleep mode exposes these phases to the scheduler and enables their acceleration on the fast core. Applications on the left side of the chart, however, experience performance *degradation*. Those with the highest overhead (RNA, `equake` and `applu`) run frequent short parallel loops. Despite being well-balanced applications, the asymmetry of the platform causes the thread running on a fast core to complete its share of these loops earlier. In this case, the sleep mode makes the fast core become idle very often, triggering frequent migrations that introduce substantial overheads. An adaptive mode alleviates this issue, but the blocking threshold must be sufficiently large to remove the overheads completely.

Figure 5.3 shows how the fraction of time spent in sequential phases as seen by the OS changes for different blocking thresholds. This further underscores that the blocking threshold for the adaptive mode must be chosen carefully: choosing a very large threshold reduces the visibility of sequential phases for the OS, but a small one causes overhead, as shown in Figure 5.2.

We now evaluate the PA algorithm with Runtime Extensions (PA-RTX). We test the *spin-then-notify* synchronization mode using several spin thresholds. The blocking threshold is set at 100m iterations in all experiments. We also test the feature permitting the application to expose the *master thread*. These scenarios are compared with the Default scheduler where applications use the adaptive mode, and with the base PA algorithm (no RTX) using the spin mode (PA-base (spin)) and the adaptive mode (PA-base (adaptive)). For PA-base (adaptive) we used the best blocking thresholds: 10m and 1m respectively. Figure 5.4 shows the results.

Overall, we conclude that PA-RTX is less sensitive to the choice of thresholds than PA-base (adaptive). PA-base (adaptive) degrades the performance for several applications, up to a maximum of as much as 26%(!) when a low value of the blocking threshold is used. PA-RTX degrades the performance by 4% at the most and only in one case, and that happens when the spin threshold is set to an extremely small

value of 1K iterations. With adaptive synchronization, a trade-off must be made when setting the blocking threshold: choosing a small value may lead to degrading the performance, but choosing a value that is too high will hide sequential phases to the scheduler. With the *spin-then-notify* primitive, choosing the right threshold is much easier: the spin threshold can be safely set at a low value of several hundred thousand or a million iterations, and the blocking threshold can be set at a very high value to avoid performance loss. Because of this flexibility, PA-RTX even outperforms PA-base (adaptive) with the best threshold, by as much as 5% in some experiments.

5.2.3. Multi-application workloads

Figure 5.7 shows that even simple asymmetry-aware schedulers trivially accelerate sequential phases and beat the default scheduler when there is only one parallel application running in the system. But as we demonstrate next, they fail to achieve improvements comparable to PA in more realistic multi-application scenarios.

This section shows our results for multi-application workloads. We study the performance of RR, BusyFCs, Static-IPC-Driven, Min-TLP and PA and compare it with Default on the 1FC-12SC and 4FC-12SC configurations. Runtime extensions are not used in this case, and the applications run under the adaptive synchronization mode with the default blocking threshold. Tables 5.3 and 5.4 show the two sets of multi-application workloads we used for our evaluation. The workload names in the left column of both tables indicate the class of each application listed in the same order as the corresponding benchmarks, so for example in the STCI-PSMI category `gamess` is the single-threaded compute-intensive (STCI) application and `FFTW` is the partially sequential memory-intensive (PSMI) application. Note that all highly parallel applications have been presented as "HP" without distinction between memory- and CPU-intensive subclasses. Their MI/CI suffix has been removed deliberately to emphasize that schedulers that rely on the number of active threads when making scheduling decisions (Min-TLP and PA) map all threads of HP applications on slow cores regardless of their memory-intensity (either HPCI or HPMI). Nevertheless, the class of each HP application in the workloads can be found in Table 5.2.

The numbers in parentheses next to each parallel application in Tables 5.3 and 5.4 indicate the number of threads chosen for that application. In the first set, the first number corresponds to the 1FC-12SC configuration and the second one to the 4FC-12SC configuration. The workloads from the second set (Table 5.4) were run on the 4FC-12SC configuration only, so one thread number is included next to each parallel application.

Figures 5.5 and 5.6 show the results for workload set #1 on 1FC-12SC and workload set #2 on 4FC-12SC, respectively. Results for workload set #1 on 4FC-12SC were quantitatively similar, so we opted to omit the associated graph. In each graph, there is a speedup bar for each application in the workload as well as the geometric mean speedup for the workload as a whole, labeled with the name of the workload

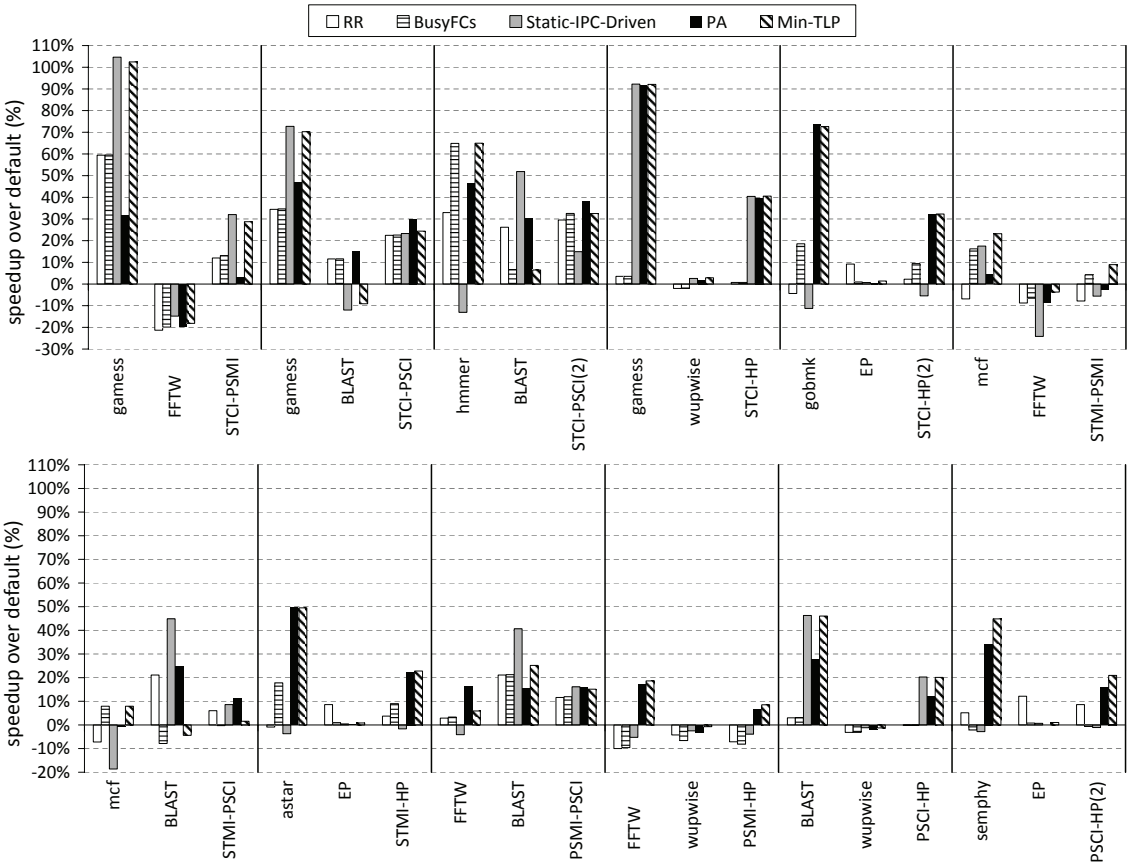


Figure 5.5: Speedup of asymmetry-aware schedulers on 1FC-12SC for multi-application workload set #1.

from Tables 5.3 and 5.4. We provide a detailed discussion of performance on the 1FC-12SC scenario (results for the 4FC-12SC configuration are interpreted with similar explanations).

Examining performance results of the simple asymmetry-aware schedulers – BusyFCs and RR– in Figure 5.5, we see that these algorithms deliver non-negligible speedups over Default for workloads with a low number of active threads (e.g., pairs consisting of an ST and a PS application, or two PS applications). When the number of threads is small, the probability that these schedulers map the "most suitable" thread to the fast core is rather high, so their performance is close to more sophisticated schedulers. In cases where HP applications are present, however, PA and MinTLP significantly outperform these simpler schedulers, delivering up to 40% performance improvements (STCI-HP) over them as well as over Default.

MinTLP and PA offer different performance in some cases. Although PA is fairer, because it shares fast cores equally among eligible threads, fairness sometimes comes at a cost. This is especially evident in scenarios with STCI-PSMI and STMI-PSMI workloads, where PA fair-shares the fast core between the ST application and the thread running sequential phase of the PSMI application. Since the PS application is memory-intensive, constantly migrating its *serial* thread between fast and slow cores, which *do not* share a last-level cache in this configuration, degrades

the performance. At the same time, in workloads consisting of a single-threaded application and a partially sequential compute-intensive application (STCI-PSCI, STCI-PSCI(2) and STMI-PSCI), fair-sharing the fast core enables PA to deliver fairness and even outperform Min-TLP.

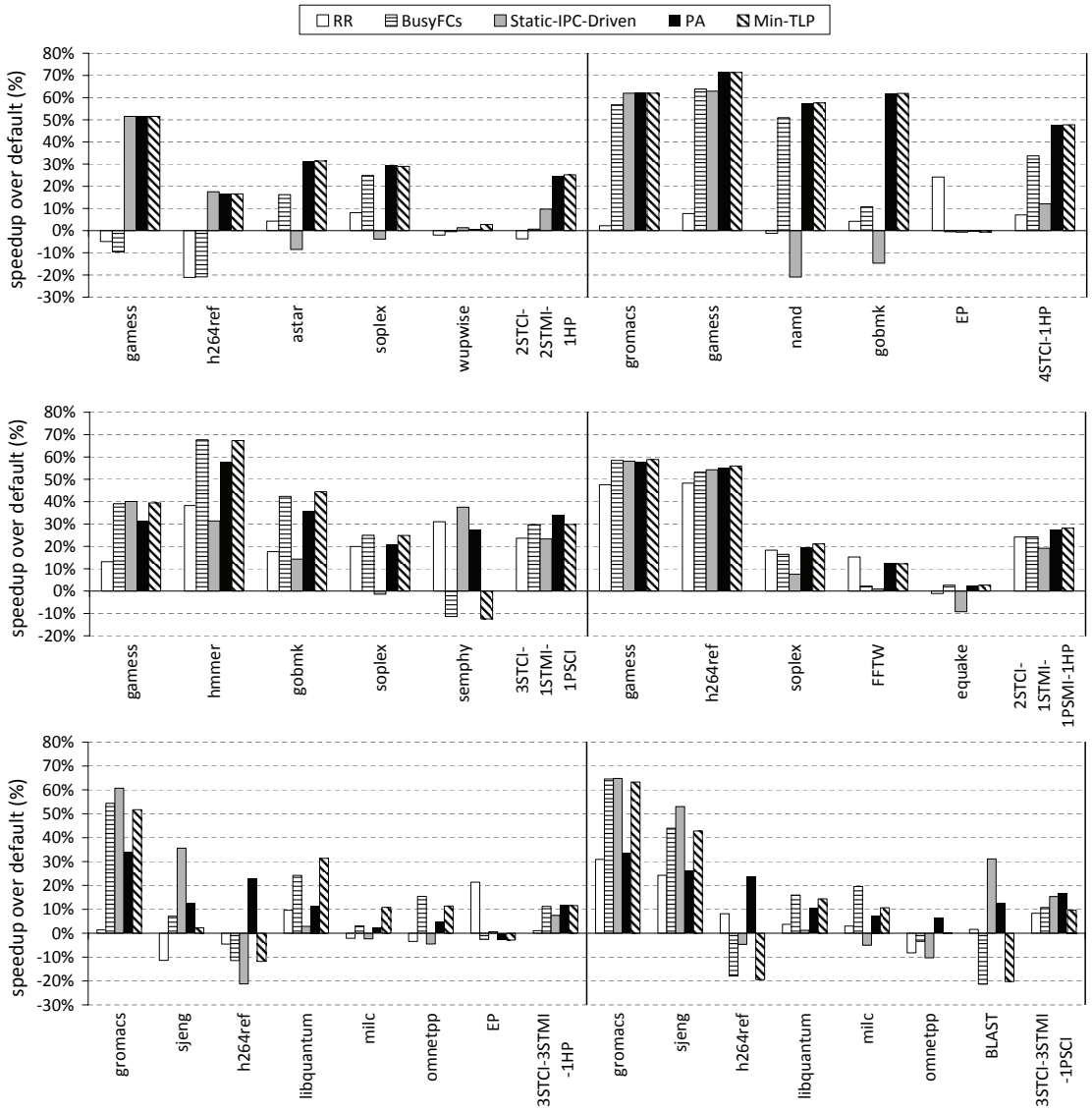


Figure 5.6: Speedup of asymmetry-aware schedulers on 4FC-12SC for multi-application workload set #2.

The implication of these results is that migration overhead must be reduced or taken into account if a scheduler is to deliver both performance and fairness. This is addressed by a *migration-friendly* system configuration and a *topology-aware* scheduler design, as will be explained in Section 5.2.4. Figure 5.6, for the 4FC-12SC migration-friendly topology, demonstrates this scenario. In this case, many of the costly migrations of threads across memory hierarchy domains (i.e., among cores that do not share a last-level cache) are eliminated, and so PA almost never under-performs MinTLP for the workload as a whole.

We now turn our attention to the Static-IPC-Driven algorithm. Overall, we see

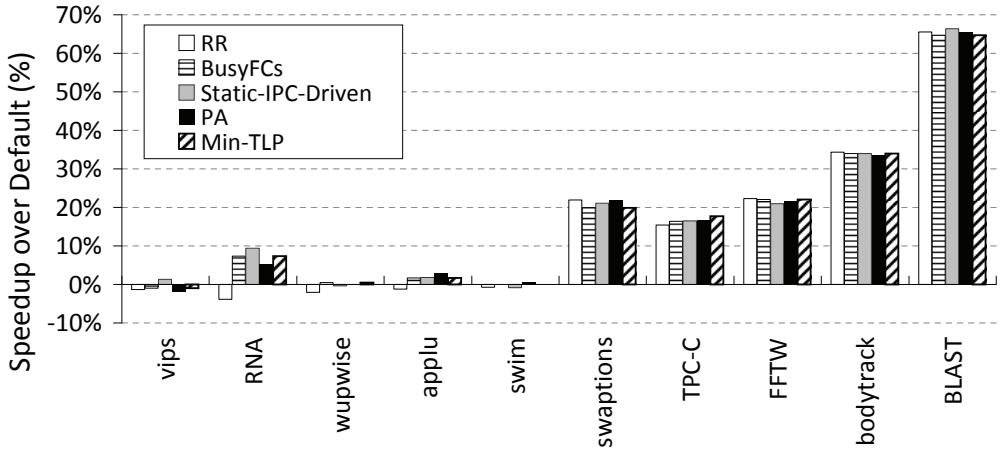


Figure 5.7: Speedup for single-applications workloads on 4FC-12SC.

that this algorithm performs comparably to PA and MinTLP only in some workloads including an STCI application. In these cases, Static-IPC-Driven assigns the ST thread to the fast core, because it has the highest static IPC ratio due to its compute-bound nature. But because this thread also happens to be the most suitable candidate from the thread-level parallelism standpoint, Static-IPC-Driven makes a decision similar to MinTLP and PA. However, in cases where the least suitable application (HP) is more compute bound, e.g., EP in the STCI-HP(2) workload, Static-IPC-driven makes a wrong decision and performs worse than PA and MinTLP. For the other workloads, PA and MinTLP outperform Static-IPC-Driven. These results once again demonstrate that only parallelism-aware schedulers perform well across the board for a wide range of workloads.

5.2.4. Evaluating the overhead

Cross-core migrations are an essential mechanism in any asymmetry-aware scheduler. Migrations can be especially costly if the source core is in a different domain of the memory hierarchy than the target core (i.e., the two cores do not share a last level cache). Migration cost is defined as the migration-induced performance loss relative to the Default scheduler. To measure only the performance loss and not the performance improvements resulting from the asymmetry-aware policy, we set all the cores in the system to be slow. The asymmetry-aware scheduler, however, still "thinks" that some cores are fast, so it still performs the migrations in accordance with its policies, incurring the cost but not reaping the benefit.

We hypothesized that migration overhead would be mitigated on systems with a *migration-friendly* topology: where at least one fast core would be in the same memory-hierarchy domain with several slow cores, and where the scheduler would avoid cross-domain migrations when possible (*topology aware*). To evaluate this hypothesis, we study the overhead of PA on several different configurations: 1FC-12SC, 1FC-3SC, and 4FC-12SC. In the first configuration, the fast core is in a separate memory hierarchy domain than the slow cores. This is *not* a migration-

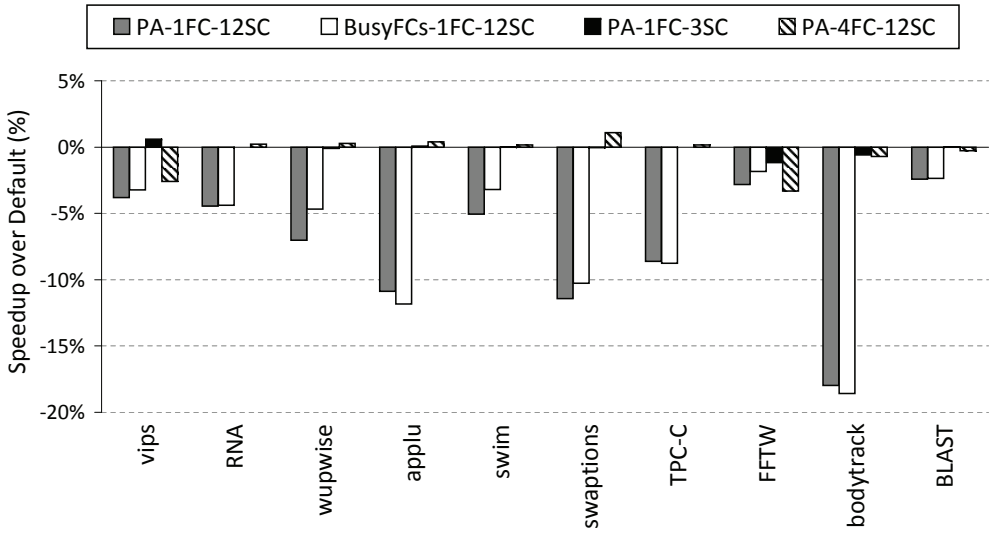


Figure 5.8: Migration overhead.

friendly topology. In the 1FC-3SC configuration only one memory hierarchy domain is used and all cores, fast and slow, are within that domain. This is the most migration-friendly topology. Finally, 4FC-12SC is a hybrid configuration where each fast core shares a memory hierarchy domain with three slow cores. This topology is complemented by policies that avoid cross-domain migrations whenever possible[7], and so this configuration is also migration-friendly.

Figure 5.8 shows the overhead of migrations with the PA algorithm on these three configurations. We use a set of parallel applications exposing a wide variety of sequential portion and memory intensity. The numbers show the relative speedup over Default, so lower numbers mean higher overhead. We also show the overhead for BusyFCs on 1FC-12SC, to demonstrate that migration overhead is fundamental to asymmetry-aware schedulers in general, not only to PA and MinTLP.

We observe that the highest overheads are incurred on a migration-unfriendly 1FC-12SC topology. On a migration-friendly 1FC-3SC topology, the overheads become negligible. On 4FC-12SC, a more realistic configuration, the migration overheads are also very low.

Our conclusion is that asymmetry-aware policies can be implemented with relatively low overhead on AMP systems, even when it is not possible to ensure that all cores share a single memory hierarchy domain. The key is to design the system such that fast cores share a memory domain with at least some slow cores and, very importantly, to extend the scheduler to avoid cross-domain migrations when possible.

[7]PA, MinTLP as well as our implementations of RR, BusyFCs and Static-IPC-Driven, are topology-aware.

5.3. Related work

Scheduling for AMP systems is a rather new research area, but there have already been several other asymmetry-aware scheduling algorithms proposed in earlier work. The key difference of most of that earlier work from ours is that it focused primarily on algorithms whose goal was to optimize performance on AMPs by catering to microarchitectural diversity of the workload. In contrast, our work caters to the diversity in the application-level parallelism. The aim of this section is to lay out some of the background on previously proposed scheduling algorithms that addressed this goal.

We begin by illustrating why the exploitation of microarchitectural diversity alone is not sufficient to fully unleash the potential of AMP systems. In Section 4.4, we introduced some of the previously proposed algorithms exploiting the workload's microarchitectural diversity to improve efficiency on AMPs. On the high level, they all rely on monitoring speedup of individual threads on fast cores relative to slow cores and map to fast cores those threads that experienced the largest relative speedup [3, 2]. Because these algorithms do not take into account application-level parallelism, they could easily map to a fast core a thread of a highly parallel application while leaving threads running sequential applications on slow cores. Consider a simple example demonstrating the potential shortcomings of failing to consider application-level parallelism in an algorithm similar to Kumar's [3]. Suppose we have an AMP processor with one fast core and nine slow cores and two applications: a scalable parallel application with nine threads and a sequential application with a single thread. Suppose further that the parallel application runs CPU-intensive code and that each thread speeds up by a factor of two on the fast core relative to a slow core. The sequential application, on the other hand, runs memory-intensive code, so it only speeds up by a factor of 1.5× on the fast core[8]. Kumar's algorithm would thus map a thread of a parallel application to run on the fast core, but since only one thread will be able to run on the fast core at a time, the entire application would speed up (optimistically) by a factor of 1.1× relative to running on slow cores the entire time. The sequential application would experience no speedup (a speedup factor of 1×), so the average speedup for the workload would be 1.05×, or 5% relative to running the entire workload on slow cores. If, on the other hand, the scheduler considered application-level parallelism and mapped the thread of the sequential application to the fast core, the entire workload would enjoy the speedup factor of 1.25× (the sequential application would speedup by 1.5× and the parallel one by 1×). The resulting workload-average performance improvement would be 25% – five times better than with the algorithm that does not consider application-level parallelism. Our PA scheduler does consider application-level parallelism in scheduling decisions, and so it will not miss on the performance improvement opportunities that would evade the schedulers proposed by Kumar or Becchi.

We now switch the discussion to the related work that focused on leveraging AMPs to accelerate serial phases of parallel applications. Hill and Marty [5] and Morad,

[8]These speedup factors are comparable to those reported in Chapter 4.

Weiser and Kolody [23] derived theoretical models for speedup on AMPs for parallel applications with serial phases. These models assumed a scheduler like PA which maps serial phases to fast cores. The former group concluded that AMPs have the potential to offer performance significantly better than symmetric multicore processors for applications whose serial phases constituted at least 5% of the execution time. This theoretical work nicely articulated the potential of AMPs combined with a PA policy, but addressing design of an OS scheduler and evaluating it on a real system was outside the scope of that work. Furthermore, these theoretical models understandably explored only a limited subset of workloads, and did not analyze the important scenario of multi-application workloads. Our implementation permits us to address the limitations of this work and perform a thorough evaluation of the PA policy on real hardware for a range of real applications and multi-application workloads.

Another related study was performed by Annavaram et al [4]. This study was experimental: in it, the authors modified several parallel applications to run their serial phases on a fast core. This study was interesting in that it allowed to validate prior theoretical analysis on real applications. But the main shortcoming relative to our work is that the authors did not pursue the design of a real OS scheduler. Applications were modified manually to schedule themselves on an AMP, and while this was entirely appropriate for the goals of that study, the manual approach is not a good general solution, because it places undue burden on application developers and would work poorly for multi-application workloads. If each application in a multi-application workload mapped its serial phase to a fast core without the knowledge that other applications had done the same, fast cores could become overloaded relative to slow cores and the threads mapped to fast cores could experience long wait times in the CPU run queue. This would result in serial phases running *slower* than if they had been left to run on slow cores. To prevent this from occurring, there has to be a global arbiter that would equally share fast cores among threads desiring to run on them and that would periodically map those threads to slow cores in scenarios where load imbalance may occur. Our scheduler provides the desired functionality.

ACS is a system that was explicitly designed to accelerate lock-protected critical sections on AMPs [80]. ACS uses a combination of compiler and hardware techniques to ensure that the code inside the critical section is executed on the fast core. Thread migrations are performed entirely in hardware, so the design of an OS scheduler was not addressed. Furthermore, like Annavaram's work, the system supports only single-application workloads. The authors indicate that operating system assistance would be required to support multi-application workloads. Our work provides support similar to that which would be needed by ACS.

Finally, we discuss several simpler asymmetry aware scheduling algorithms that focused on keeping the fast core busy or aimed to fairly share fast and slow cores among threads. Two algorithms proposed by Li [17] and Balakrishnan [14] had the goal of ensuring that fast cores never go idle before slow cores, akin to BusyFCs. Prior work also discussed a simple scheduler for AMP that shared fast and slow cores among threads in a round-robin fashion [2, 81]. For the evaluation shown in

this thesis, we created an implementation of the two AMP schedulers just discussed (the round-robin one and the one that keeps fast cores busy), and we compared them to PA. We found that these schedulers effectively accelerate serial phases for a trivial case of single-application workloads, as long as unused threads are put to sleep during the serial phase. Nevertheless, our experimental results also revealed that in multi-application scenarios or in the event that unused thread of parallel applications spin rather than blocking, these simple schedulers do not accomplish this task (as shown in Section 5.2).

Another work relevant to our research is on Feedback-Driven Threading (FDT) by Suleman, Qureshi and Patt [35]. In this work, the authors developed a runtime system that performs an online discovery of the optimal threading level for parallel applications. Of particular relevance to us is the discussion provided by the authors about the sources of scalability bottlenecks in parallel applications. Our scheduler addresses those bottlenecks that are due to load imbalance or data serialization. The authors also identify memory bandwidth as another possible bottleneck. With this type of bottleneck, there is not a well defined serial phase, but the wait times due to memory bus contention are arbitrarily distributed among the threads. In this case, AMP systems coupled with PA policy are less appropriate to mitigate serial bottlenecks, because there is not a well-defined serial phase that can be accelerated on a fast core. To address these types of bottlenecks, application-level solutions such as the aforementioned FDT are more suitable and effective.

5.4. Conclusions

In this chapter we have analyzed the benefits and drawbacks of parallelism-aware scheduling policies for AMP systems. While some algorithms that do not consider TLP perform comparably to the proposed algorithms PA and MinTLP in some scenarios, none of them perform well across the board. PA and MinTLP outperform the other schedulers by as much as 40% in some cases. This indicates the importance of considering the TLP (thread-level parallelism) in asymmetry-aware scheduling. We have also proposed a small set of runtime extensions to complement the OS scheduler and reduce the possibility of wastefully scheduling busy-waiting threads on fast cores.

Our results have shown that PA and MinTLP effectively schedule workloads including parallel applications. However, an asymmetry-aware algorithm that relies on relative speedup, such as HASS (presented in Chapter 4), is able to outperform parallelism-aware scheduling policies for workloads are consisted of single-threaded applications only. This fact further underscores that a comprehensive scheduler for asymmetric multicore systems should cater to both the TLP and relative speedup to to be able to maximize system-wide performance on asymmetric systems regardless of the nature of the application mix. This is the main focus of the next chapter.

Chapter 6

A Comprehensive Scheduler for Asymmetric Multicore Systems

In previous chapters, we demonstrated that *specializing* each core type for applications that will use it most efficiently is the key to deliver the potential benefits of AMPs to unmodified applications. Specialization must be aided by a thread scheduler that decides which threads to run on fast cores and which ones on slow cores.

Two kinds of operating system schedulers emerged to address this challenge. The first type targeted ILP specialization, among which HASS is included, by assigning the most CPU-intensive threads to fast cores (as shown in Chapter 4). The second type targeted TLP specialization by assigning sequential applications and sequential phases of parallel applications to run on fast cores; the PA algorithm described in Chapter 5 is an example of the latter type. Both types of schedulers have proved beneficial for their respective target workloads: ILP specialization delivered benefits for workloads consisting of single-threaded applications, and TLP specialization was proved effective for workloads where parallel applications are present. It was not made clear, however, whether it is worth combining these two approaches into a single algorithm and what would be the impact of this comprehensive scheduling solution. In other words, should general-purpose operating systems for asymmetric processors use an algorithm focusing on ILP specialization, an algorithm focusing on TLP specialization, or an algorithm that performs both types of specialization?

The aim of this chapter is to shed light on this question. To that end, we propose CAMP, a new Comprehensive scheduling algorithm for AMP systems that delivers both types of specialization. To the best of our knowledge, this is the first asymmetry-aware algorithm addressing this goal. The challenge in implementing this algorithm is equipping it with an effective mechanism for deciding which threads are more "profitable" candidates for running on fast cores. To make that possible, we introduce the new metric *Utility Factor* (UF), which accounts for both the ILP and TLP of the application and produces a single value that approximates how much the application as a whole will improve its performance if its threads are allowed to occupy all the fast cores available on that system. The utility factor is de-

signed to help the scheduler pick the best threads to run on fast cores in non-trivial cases, such as the following. Consider a workload consisting of a CPU-intensive application with two runnable threads and a less CPU-intensive one with a single thread. In this case, it is not immediately clear which thread is the best candidate for running on the fast core (assuming there is only one fast core on the system). On the one hand, dedicating the fast core to a thread of a two-threaded application may bring smaller performance improvements to the application as a whole than dedicating the fast core to the single-threaded application, because a smaller part of the application will be running on fast cores in the former case. On the other hand, the two-threaded application is more CPU-intensive, so running it on the fast core may be more profitable than dedicating the fast core to another, less CPU-intensive application. By comparing utility factors across threads the scheduler is able to identify the most profitable candidates for running on fast cores.

In order to account for the relative ILP component of a thread's utility factor, CAMP must determine the *speedup factor* that the thread experiences from running on a fast core relative to a slow core. In Chapter 4, we showed that previous online approaches for determining the speedup factor posed serious limitations. This led us to devising a new online method based on estimation, which does not require to run each thread on cores of both types (see Section 4.1.3). CAMP relies on this very same method. While the method does not approximate the speedup factor with a high accuracy, since the model was made deliberately simple for portability across hardware platforms, it rather successfully categorizes applications into three classes – low, medium and high – according to their efficiency. As a result, the scheduler using dynamically estimated speedup factors performs within 1% of the oracular scheduler that uses *a priori* known, and thus highly accurate, overall speedup factors.

As the other asymmetry-aware algorithms proposed in this thesis, CAMP was implemented in the OpenSolaris operating system and evaluated on real multicore hardware where asymmetry is emulated by setting the cores to run at different frequencies via DVFS. We compare CAMP with several other asymmetry-aware schedulers including the proposed Parallelism-Aware (PA) scheduler, which performs only TLP specialization; our implementation of a Speedup-Factor Driven (SFD) scheduler, which only caters to ILP specialization; and a baseline round-robin (RR) scheduler that simply shares fast cores equally among all threads. We found that for workloads consisting exclusively of single-threaded applications, the algorithm focusing only on ILP specialization is sufficient, but this algorithm is ineffective for workloads containing parallel applications. Conversely, an algorithm focusing only on TLP specialization is effective for workloads containing parallel applications, but not for those where only single-threaded applications are present. CAMP, on the other hand, effectively addresses both types of workloads. We also found that there is some extra benefit in using information on ILP in addition to TLP for realistic workloads containing parallel applications. The greatest benefit of CAMP, therefore, is that it optimizes scheduling on AMPs for a variety of workloads, smoothly adjusting its strategy depending on the type of applications running on the system.

The rest of the chapter is organized as follows. Section 6.1 describes how we compute the utility factor. Section 6.2 presents the design of the CAMP algorithm and briefly describes other algorithms that we use for comparison. Section 6.3 describes the experimental results. Section 6.4 discusses related work. Finally, Section 6.5 summarizes our findings.

6.1. Utility Factor

Given a system with *NFC* fast cores, the *Utility Factor* (*UF*) is a metric approximating the application speedup if *NFC* of its threads are placed on fast cores and any remaining threads are placed on slow cores, relative to placing all its threads on slow cores. By convention, the speedup is measured using the following formula: $Speedup = T_{base}/T_{alt}$, where T_{base} is the completion time for the application in the "base" configuration, where only slow cores are used, and T_{alt}, is the completion time in our "alternative" configuration, where both fast and slow cores are used.

The formula for the utility factor was obtained using an analytical approach. Essentially, we begin by deriving a formula for the theoretical speedup that a perfectly balanced parallel application (i.e., a parallel application with negligible serial bottlenecks) obtains from a scheduler that keeps fast cores busy, with respect to an execution where only slow cores are used. Note that such a scheduling policy was already presented in the previous chapter as the BusyFCs scheduler. Whenever a thread blocks or exits while running on a fast core, BusyFCs immediately picks another thread running on a slow core and migrates it onto the idle fast core, thus maximizing its utilization. A thorough analysis of the theoretical performance of this scheduler enables us to estimate the potential benefits that come from mapping some of the threads of a parallel application to fast cores. Notably, this analysis makes it possible to identify situations where little performance can be expected from using fast cores.

We found that the formula for BusyFCs's theoretical speedup turned out to be far too complex to be computed and repeatedly updated at runtime by the OS. For the sake of efficiency, we opted to derive a simpler formula that approximates the first one rather accurately, and at the same time, enables the CAMP scheduler to dynamically, and in a lightweight fashion, classify applications based their on TLP and ILP characteristics.

The remainder of this section is organized as follows. Section 6.1.1 illustrates the derivation process that leads to the formula for the theoretical speedup of BusyFCs. Section 6.1.2 introduces a simpler formula that the CAMP scheduler uses at runtime to efficiently approximate such theoretical speedup.

6.1.1. Theoretical speedup of BusyFCs

In constructing the model for the theoretical speedup we make four simplifying assumptions:

- Only the threads of the target application (i.e., the application for which the speedup is estimated) are allowed to use fast cores. This would not be the case under a fair scheduling policy, which would attempt to share fast cores among all "eligible" threads from different applications.
- The number of threads in the application does not exceed the number of slow cores. Given that the number of slow cores is likely to be large relative to fast cores, this assumption is, first of all, reasonable, because CPU-bound applications are not likely to be run with more threads than cores [26]; and, second, it will not introduce a significant error into our model, at least for the applications that we considered (parallel scientific applications from SPEC OMP2001 suite, plus a few others). As the number of threads begins to exceed the number of fast cores, the speedup rapidly decreases to zero. So even if the application has more threads than assumed by our formula, the speedup should remain accurate.
- For parallel scientific applications, such as the ones explored in this thesis, we observed that threads in the application exhibit very close speedup factors. This stems from the fact that threads in these applications usually execute the same code with different data. As a result, we opted to use the *average* speedup factor of all the threads application (SF_{avg}) to approximate the SF of *any* thread in it.
- The theoretical speedup derived in this section does not account for migration overhead. As a result, the formula approximates an upper bound of the achievable speedup.

We also make several assumptions about the nature of the application. First, we assume that the application is *perfectly balanced*, namely, all its threads perform the same amount of work in parallel until completion. Second, the application consists of k balanced parallel sections, $k \geqslant 1$, separated by global synchronization points (e.g., barriers). Within a parallel section P, all threads complete the same number of instructions (NI_P) in parallel (the number of completed instructions may differ across parallel sections, though). The latter assumption indirectly states that all threads will execute the same number of instructions till completion. More precisely, if NI denotes the total number of instruction executed by each thread then $NI = \sum_{i=1}^{k} NI_i$.

The completion time of a parallel application is determined by the *slowest* thread[1]. More specifically, determining the completion time of the application entails esti-

[1]In this chapter, we focus on studying the performance of scientific applications rather than transactional applications such as databases or web servers. For that reason, we use the wall clock

mating the execution time of the slowest thread for each parallel section.

In any execution where only slow cores are used, all threads will take roughly the same time to complete all the instructions associated to a given parallel section. (Recall that the application is perfectly balanced, so any thread could be considered as the *slowest* one.) As a result, we approximate this completion time (CT_{slow}) with the time that any thread with speedup factor SF_{avg} takes to complete NI instructions on a slow core.

Determining the execution time of the slowest thread in a parallel section under BusyFCs ($CT_{BusyFCs}$) is not so straightforward. To explain how we approximate this value, we rely on some additional notation:

- NFC, NSC : Number of fast and slow cores of the AMP system, respectively.
- $N_{threads}$: Total number of threads the parallel application runs with.
- NI_{fc}, NI_{sc} The total number of instructions that the *slowest* thread completes on fast and slow cores under BusyFCs, respectively. Note that $NI = NI_{fc} + NI_{sc}$.
- $NI_{fc,i}, NI_{sc,i}$ The total number of instructions that the *slowest* thread completes on fast and slow cores in the i-th parallel section, respectively. Note that $NI_i = NI_{fc,i} + NI_{sc,i}$.
- SPI_{fc}, SPI_{sc} : Average number of seconds per instruction[2] on fast and slow cores, respectively.

The initial formulas for $CT_{BusyFCs}$, CT_{slow} and SF_{avg} are as follows:

$$\begin{aligned} CT_{BusyFCs} &= NI_{fc} * SPI_{fc} + NI_{sc} * SPI_{sc} \\ &= \sum_{i=1}^{k} (NI_{fc,i} * SPI_{fc} + NI_{sc,i} * SPI_{sc}) \end{aligned} \tag{6.1}$$

$$\begin{aligned} CT_{slow} &= NI * SPI_{sc} \\ &= \sum_{i=1}^{k} (NI_i * SPI_{sc}) \end{aligned} \tag{6.2}$$

$$SF_{avg} = \frac{SPI_{sc}}{SPI_{fc}} \tag{6.3}$$

(or completion) time of the application to assess its performance rather than throughput-oriented metrics, such as number of transactions per time unit.

[2]Note that we opted to use the SPI rather than other, more widely used performance metrics such as the number of cycles per instruction (CPI) or instructions per second (IPS). We found that using the SPI in the derivation process shown later in this section makes it possible to simplify calculations considerably.

Algorithm 6.1 Execution of a parallel phase under BusyFCs

Definitions: R is the set of runnable threads in the application. NFC is the number of fast cores on the AMP.
while $R \neq \emptyset$ **do**
  $f \leftarrow min(NFC, |R|)$
  Remove f threads from R and map/migrate them to fast cores
  Ensure that threads in R run on slow cores
  Wait until thread in fast cores complete (fast cores become idle)
end while

The execution time of the slowest thread in a parallel section under BusyFCs can be obtained from the fraction of instructions executed by this thread on fast and slow cores. Prior to describing how those fractions can be computed, we describe the set of actions performed by the BusyFCs scheduler during the execution of a given parallel phase P, where each thread has to execute NI_P instructions to reach the synchronization point at the end of the phase. At the beginning of the parallel section, BusyFCs maps as many threads as possible to fast cores (NFC threads at the most), while the remaining ones are mapped to slow cores. Threads on fast cores complete all their instructions associated to this parallel phase, NI_P, and then block, leaving fast cores idle as a result. Conversely, threads on slow cores complete $\frac{NI_P}{SF_{avg}}$ instructions during this period. As soon as fast cores become idle, BusyFCs migrates the *next* NFC threads to fast cores. The fast cores will become idle again as soon as these threads complete their remaining instructions. This sequence of actions is summarized in Algorithm 6.1. In each iteration of the algorithm, BusyFCs migrates as many threads as possible to fast cores and waits until these threads complete. Since each iteration NFC threads are migrated onto fast cores, there will be $\lfloor \frac{N_{threads}-1}{NFC} \rfloor + 1$ iterations[3].

Table 6.1 shows F_{fc} and F_{sc}, the fraction of instructions executed on fast and slow cores respectively. Each row of the table displays the values for a specific iteration, $F_{fc}(i)$ and $F_{sc}(i)$. In the first iteration, threads running on fast cores complete all their instructions NI_P, while threads on slow cores complete as many as $\frac{NI_P}{SF_{avg}}$ instructions. Equivalently, these values can be expressed in terms of the fraction over NI_P, as 1 (100%) and $\frac{1}{SF_{avg}}$, respectively. In the second iteration, threads running on fast cores have to complete a fraction of $1 - \frac{1}{SF_{avg}}$. This works out that way because those threads already completed $\frac{1}{SF}$ on slow cores in the previous iteration, so they will run on fast cores until they complete their remaining fraction: $1 - \frac{1}{SF_{avg}}$.

According to the information displayed in Table 6.1, the fraction of instructions executed by threads mapped to fast and slow cores in the iteration i, denoted by

[3]The function $\lfloor x \rfloor$ denotes the integer part of x. In other words, $\lfloor x \rfloor$ is the largest integer not greater than x.

Table 6.1: Fraction of the number of instructions executed on fast and slow cores when threads of the parallel application run under the BusyFCs scheduler

Iteration	$F_{fc}(i)$	$F_{sc}(i)$
#1	1	$\frac{1}{SF_{avg}}$
#2	$1 - \frac{1}{SF_{avg}}$	$\frac{1}{SF_{avg}} + \frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}$
#3	$1 - \left(\frac{1}{SF_{avg}} + \frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}\right)$	$\frac{1}{SF_{avg}} + \frac{1-\frac{1}{SF_{avg}}}{SF_{avg}} + \frac{1-\frac{1}{SF_{avg}}+\frac{1-\frac{1}{SF_{avg}}}{SF_{avg}}}{SF_{avg}}$
...	...	...
#i	$1 - F_{sc}(i-1)$	$F_{sc}(i-1) + \frac{F_{fc}(i)}{SF_{avg}}$

$F_{fc}(i)$ and $F_{sc}(i)$, can be defined mutually recursively as follows:

$$F_{fc}(1) = 1 \tag{6.4}$$

$$F_{sc}(1) = \frac{1}{SF_{avg}} \tag{6.5}$$

$$F_{fc}(i) = 1 - F_{sc}(i-1) \tag{6.6}$$

$$F_{sc}(i) = F_{sc}(i-1) + \frac{F_{fc}(i)}{SF_{avg}} \tag{6.7}$$

Simplifying the previous equations we have that:

$$F_{fc}(1) = 1 \tag{6.8}$$

$$F_{fc}(i) = F_{fc}(i-1) - \frac{F_{fc}(i-1)}{SF_{avg}} = F_{fc}(i-1) * (1 - \frac{1}{SF_{avg}}) \tag{6.9}$$

Furthermore, we can still unroll this equation and obtain a non-recursive formula:

$$\begin{aligned} F_{fc}(i) &= F_{fc}(i-1) * (1 - \frac{1}{SF_{avg}}) \\ &= F_{fc}(i-2) * (1 - \frac{1}{SF_{avg}}) * (1 - \frac{1}{SF_{avg}}) \\ &= F_{fc}(1) * \underbrace{(1 - \frac{1}{SF_{avg}}) * (1 - \frac{1}{SF_{avg}}) * \cdots * (1 - \frac{1}{SF_{avg}})}_{i-1} \\ &= 1 * \underbrace{(1 - \frac{1}{SF_{avg}}) * (1 - \frac{1}{SF_{avg}}) * \cdots * (1 - \frac{1}{SF_{avg}})}_{i-1} \\ &= (1 - \frac{1}{SF_{avg}})^{i-1} \end{aligned} \tag{6.10}$$

As stated earlier, the performance of the application is determined by the time that the slowest thread takes to execute all its instructions in each parallel phase. Under BusyFCs, the slowest thread in a given parallel phase is the one mapped

to a fast core in the last iteration of Algorithm 6.1. (As the careful reader may observe, there may be several threads running on fast cores in the last iteration, but any of them will complete roughly at the same time.) As a result, determining the completion time of a given parallel phase boils down to computing the number of instructions executed on fast and slow cores by threads mapped to fast cores in the last iteration. More precisely, let $NI_{fc,i}$ and $NI_{sc,i}$ be the number instructions executed by the slowest thread in the i-th parallel section on fast and slow cores respectively, then we have that:

$$NI_{fc,i} = NI_i * F_{fc}(N_{iter}) \quad (6.11)$$

$$NI_{sc,i} = NI_i * (1 - F_{fc}(N_{iter})) \quad (6.12)$$

$$\text{where} \quad N_{iter} = \lfloor \frac{N_{threads} - 1}{NFC} \rfloor + 1$$

Finally, we approximate the theoretical speedup of the application under BusyFCs with respect to an execution where only slow cores are used, as follows:

$$Speedup_{BusyFCs} = \frac{CT_{slow}}{CT_{BusyFCs}}$$

$$= \frac{NI * SPI_{sc}}{NI_{fc} * SPI_{fc} + NI_{sc} * SPI_{sc}}$$

$$= \frac{\sum_{i=1}^{k}(NI_i * SPI_{sc})}{\sum_{i=1}^{k}(NI_{fc,i} * SPI_{fc} + NI_{sc,i} * SPI_{sc})}$$

$$= \frac{\sum_{i=1}^{k}(NI_i * SPI_{fc} * SF_{avg})}{\sum_{i=1}^{k}(NI_{fc,i} * SPI_{fc} + NI_{sc,i} * SPI_{fc} * SF_{avg})}$$

$$= \frac{SPI_{fc} * \sum_{i=1}^{k}(NI_i * SF_{avg})}{SPI_{fc} * \sum_{i=1}^{k}(NI_{fc,i} + NI_{sc,i} * SF_{avg})}$$

$$= \frac{SPI_{fc} * \sum_{i=1}^{k}(NI_i * SF_{avg})}{SPI_{fc} * \sum_{i=1}^{k}(NI_i * F_{fc}(N_{iter}) + NI_i * (1 - F_{fc}(N_{iter})) * SF_{avg})}$$

$$= \frac{SPI_{fc} * \sum_{i=1}^{k}(NI_i) * SF_{avg}}{SPI_{fc} * \sum_{i=1}^{k}(NI_i * (F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) * SF_{avg}))}$$

$$= \frac{SPI_{fc} * \sum_{i=1}^{k}(NI_i) * SF_{avg}}{SPI_{fc} * \sum_{i=1}^{k}(NI_i) * (F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) * SF_{avg})}$$

$$= \frac{SF_{avg}}{F_{fc}(N_{iter}) + (1 - F_{fc}(N_{iter})) * SF_{avg}}$$

$$= \frac{SF_{avg}}{F_{fc}(N_{iter}) * (1 - SF_{avg}) + SF_{avg}}$$

$$= \frac{SF_{avg}}{(1 - \frac{1}{SF_{avg}})^{N_{iter}-1} * (1 - SF_{avg}) + SF_{avg}}$$

$$= \frac{SF_{avg}}{(1 - \frac{1}{SF_{avg}})^{\lfloor \frac{N_{threads}-1}{NFC} \rfloor} * (1 - SF_{avg}) + SF_{avg}}$$

Hence, the resulting formula for the theoretical speedup under BusyFCs is as follows:

$$Speedup_{BusyFCs} = \frac{SF_{avg}}{(1 - \frac{1}{SF_{avg}})^{\lfloor \frac{N_{threads}-1}{NFC} \rfloor} * (1 - SF_{avg}) + SF_{avg}} \quad (6.13)$$

6.1.2. An efficient formula for the Utility Factor

The main purpose of an estimation model for the theoretical speedup of BusyFCs is to aid the CAMP scheduler in making thread-to-core assignments. Note that if a change in the number of active threads of an application (or in its speedup factor) takes place, the scheduler will have to recompute the theoretical speedup to reflect this change by using Equation 6.13. Regrettably, such a formula is too complex[4] to be repeatedly updated at runtime by the OS, and as a result, it does not turn out suitable for scheduling.

Instead, by trading off accuracy and performance in the computation, we derived a simpler formula for the Utility Factor (UF), which approximates Equation 6.13 rather accurately. The formula for the *UF* is as follows:

$$UF = \frac{SF_{avg} - 1}{(\lfloor \frac{N_{threads}-1}{NFC} \rfloor + 1)^2} + 1 \tag{6.14}$$

Recall that $N_{threads}$ is the number of threads in the application, which is visible to the operating system, since most modern runtime environments map user threads one-to-one onto kernel threads. SF_{avg} is the average speedup factor of the application's threads when running on a fast core relative to a slow core; we describe how we obtain it at runtime in Section 6.2.1.

The CAMP scheduler relies on the UF to estimate how much performance each application in the workload would derive from using all fast cores in the system. Threads in the workload with the highest utility factor will then be assigned to run on fast cores: the higher the utility factor, the more the application benefits from using fast cores.

Let us describe the intuition behind the UF's formula. The easiest way to understand it is to first consider the case where the application has only a single thread. In this case, $UF = SF_{avg}$; in other words, the utility factor is equal to the speedup that this application would experience from running on a fast core relative to a slow core. Next, let us focus on the case when the application is multithreaded. If all threads were running on fast cores, then the entire application would achieve the speedup of SF_{avg}. In that case, the denominator is equal to one and $UF = SF_{avg}$. However, if the number of threads is *greater* than the number of fast cores, then, some of threads will be mapped to fast cores for longer periods than others (as explained in Section 6.1.1) and, as a result, the overall utility factor (speedup) will be less than SF_{avg}. To account for that, we divide $SF_{avg} - 1$ by $(\lfloor \frac{N_{threads}-1}{NFC} \rfloor + 1)^2$, the square of number of iterations of the algorithm shown in 6.1. We introduced this quadratic factor in the denominator to better approximate Equation 6.13. Given that the lowest attainable speedup is 1 (no speedup), only $SF_{avg} - 1$ is divided by $(\lfloor \frac{N_{threads}-1}{NFC} \rfloor + 1)^2$.

[4]This stems from the fact that an iterative algorithm is needed to compute the exponentiation in the denominator of the right side of Equation 6.13.

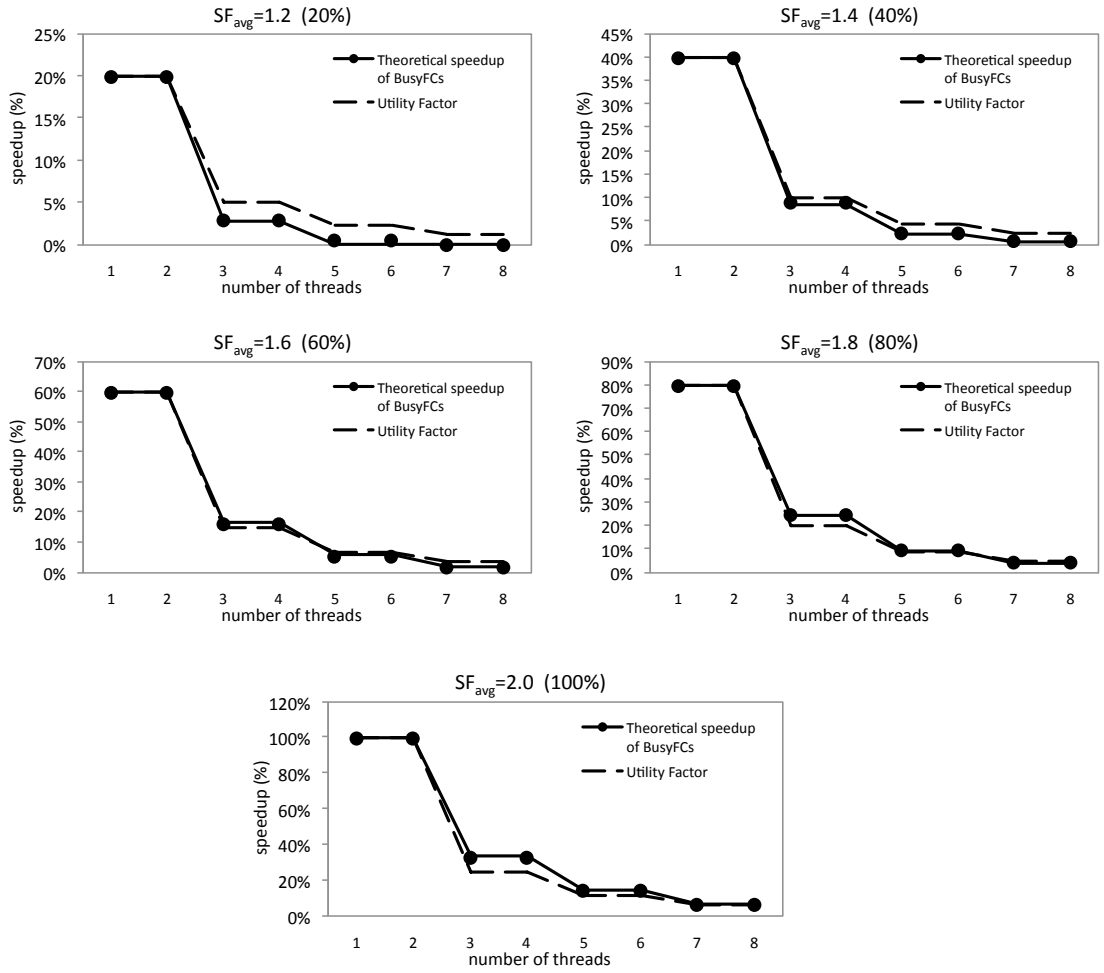


Figure 6.1: Comparison between the theoretical speedup of BusyFCs and the Utility Factor. Each graph shows the speedup obtained with both formulas on 2FC-8SCs for perfectly balanced parallel applications with different SF_{avg}s.

The formula for the utility factor provided in this section closely approximates the theoretical speedup of BusyFCs on our target asymmetric systems, where fast cores are twice as fast as slow cores (further details are provided in Section 6.3). Figure 6.1 shows the comparison between the UF and the theoretical speedup for perfectly balanced applications running on an asymmetric system with two fast cores and eight slow cores (2FC-8SC). The figure provides the comparison for hypothetical parallel applications whose SF_{avg} ranges between 1.2 and 2.0 (a typical range in this configuration).

Next, we demonstrate experimentally that the utility factor model closely approximates the actual speedup delivered by a real-world implementation of the BusyFCs scheduler. Figure 6.2 shows the estimated and actual *UF* for several highly parallel applications with different synchronization patterns and memory-intensity levels. The applications shown here belong to the SPEC OMP2001, NAS Parallel and Minebench benchmark suites. To compute the actual performance under BusyFCs, we use our own implementation of this algorithm. We performed validation for

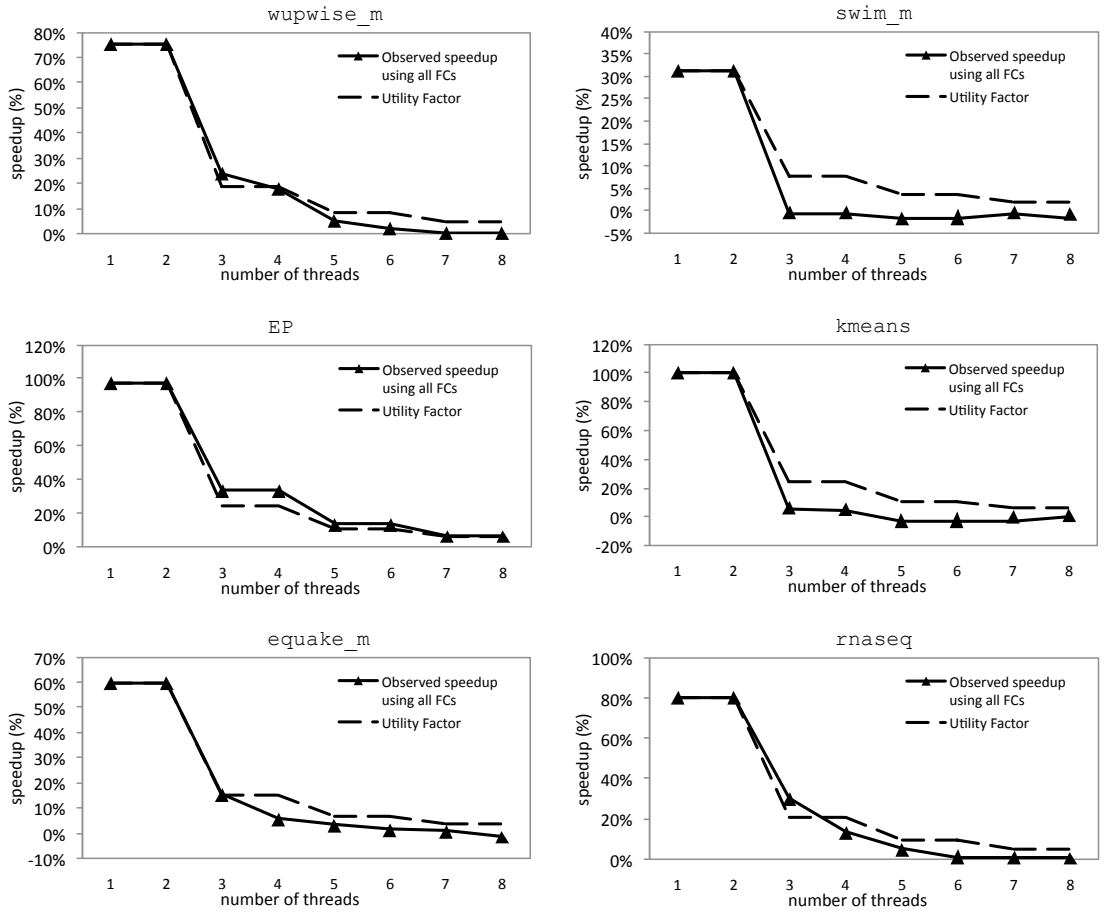


Figure 6.2: Comparison between the *UF* and the observed speedup of highly parallel applications on 2FC-8SC under BusyFCs.

other highly parallel applications from the aforementioned benchmark suites, and found that the results were quantitatively similar (so they are not reported). The actual speedup was measured on an asymmetric configuration with two fast cores and eight slow cores (2FC-8SC), which is based on the AMD-8 platform described in Section 3.3. Applications' SF_{avg}s were estimated offline by running the applications with a single thread first on slow cores, then on fast cores and computing the wall clock speedup. As evident, the estimated utility factor closely tracks the quantity it attempts to approximate.

Although we assessed the accuracy of the utility factor by focusing our study on highly parallel applications, we must underscore that the UF formula can be also used to estimate the speedup of applications with non-negligible sequential bottlenecks. In particular, we omitted a similar study for purely sequential applications, since these scenarios are equivalent to those where parallel applications run with a single thread (reported in Figure 6.2). In a similar vein, because the CAMP scheduler updates the UF as the application goes through phases with different TLP, the UF model directly works for partially-sequential applications. As pointed out in Chapter 5, those applications exhibit phases with limited parallelism where some of

their threads do useful work while the remaining ones wait, which results in phases with different number of threads and, hence different utility factor.

In this chapter, we used the utility factor to optimize *system-wide* performance. However, in cases where some applications have a higher priority than others or in scenarios where the system needs to deliver QoS guarantees to certain applications, the utility factor could be used as a *complementary* metric to find a balance between providing better service for prioritized applications and maintaining overall performance. For example, if the system determines that a high-priority application has a low utility factor, meaning that little or no speedup would be gained for that application if some of its threads were to run on fast cores, then there is no point in "wasting" a fast core on this application, despite its high priority. As a result, the utility factor would be used to ensure QoS guarantees with a minimal effect on performance.

6.2. Design and implementation

In this section, we describe the design and implementation of the CAMP scheduler as well as the other schedulers used for comparison.

6.2.1. The CAMP scheduler

The algorithm

CAMP decides which threads to place on cores of different types based on their individual utility factors. According to their utility factors, threads are categorized into three classes: LOW, MEDIUM, and HIGH. Using classes allows to mitigate some inaccuracies in estimation of the *SF* used in the *UF* formula as well as to provide comparable treatment for threads whose utility factors are very close. Threads falling in the HIGH utility class will run on fast cores. If there are more such threads than fast cores, the cores will be shared among these threads equally, using a round-robin mechanism.

If after all high-utility threads were placed on fast cores there are idle fast cores remaining, they will be used for running medium-utility threads or, if no such threads are available, low-utility threads (we do not optimize for power consumption in this work, so our scheduler tries to keep fast cores as busy as possible). In contrast with threads in the HIGH utility class, fast cores will not be shared equally for threads in the MEDIUM and LOW utility classes. Sharing the cores equally implies cross-core migrations as threads are moved between fast and slow cores. These migrations degrade the performance, especially for memory-intensive threads, because threads may lose their last-level cache state as a result of migrations. The effect of migrations on performance in asymmetry-aware schedulers was explored in Chapter 5.

Threads of parallel applications executing a sequential phase will be designated to a special class SEQUENTIAL_BOOSTED. These threads will get the highest priority for running on fast cores: this provides more opportunities to accelerate sequential phases. Only high-utility threads, however, will be assigned to the SEQUENTIAL_BOOSTED class. Medium- and low-utility threads will belong to their regular class despite running sequential phases. Since these threads do not use fast cores efficiently, it is not worthwhile to give them an elevated status. Threads placed in the SEQUENTIAL_BOOSTED class will remain there for the duration of `amp_boost_ticks`, a configurable parameter. After that, they will be downgraded to their regular class, as determined by the utility factor, to prevent them from monopolizing the fast core.

The class-based scheme followed by CAMP relies on two utility thresholds, *lower* and *upper*, which determine the boundaries between the LOW, MEDIUM and HIGH utility classes. The lower threshold is used to separate the LOW and MEDIUM classes; the upper threshold is used to separate the MEDIUM and HIGH classes.

CAMP has a built-in mechanism to dynamically, and in a transparent way, select which utility thresholds to use based on the system workload. There are two pairs of utility thresholds, one for when only single-threaded applications run on the system, and the other for when at least one multi-threaded application is running on the system. When multi-threaded applications are present, we see a higher range of utility factors than when only single-threaded applications are present, and so different thresholds than in single-threaded mode are used to reflect this higher range. These thresholds are also machine dependent: on systems with a large difference between the speed of fast and slow cores, utility factors will be larger than on systems where this difference is small.

For example, utility factors for single-threaded applications used in our study were 1.23 or higher on our experimental system (because the speedup factor relative to a slow core is at least 23%). At the same time, multi-threaded applications will often have a utility factor as low as 1 (0%), as shown in Figure 6.2. So a single-threaded application whose utility factor is low relative to other single-threaded applications will nevertheless have a high *UF* relative to most multi-threaded applications. To properly reflect the relationship between threads' *UF*s when placing them into classes, we use different sets of thresholds for single-threaded and multi-threaded scenarios.

Computing the speedup factor

For actual scheduling, CAMP must keep threads' utility factors up-to-date. To make that possible, the scheduler keeps track of changes in the number of runnable threads and detects transitions between different program phases, which may exhibit different fast-to-slow relative speedups. Keeping track of changes in the number of runnable threads is straightforward since this count is visible to the OS in most modern multithreaded environments. Obtaining accurate relative speedups (a.k.a. *speedup factors*), however, is not so straightforward.

Our method for computing the speedup factor (*SF*) relies on threads' LLC (last-level cache) miss rates measured online using hardware performance counters. The miss rate can be measured on any type of core; there is not need to run the thread on both core types. To estimate the *SF* from the LLC miss rate we used a similar approach to the one described in Section 4.1.3. On the high level, the method works as follows. We compute the hypothetical completion time for some constant number of instructions on both core types. We compose the completion time of two components: execution time and stall time. To compute the execution time we assume a constant number of instructions per cycle (machine dependent) and factor in the clock speed. To compute the stall time, we estimate the number of cycles used to service the LLC misses occurring during that instruction window: for that we require the per-instruction LLC miss rate, which the scheduler measures, and the memory latency, which can be discovered by the operating system.

This method for estimating the stall time abstracts many details of the microarchitecture: the fact that not all cache misses stall the processor because of out-of-order execution, the fact that some cache misses are actually pre-fetch requests that also do not stall the processor, the fact that some cache misses can be serviced in parallel, and the fact that the memory latency may be different depending on memory bus and controller contention as well as non-uniform memory access (NUMA) latencies on some architectures. Accounting for all these factors is difficult, because their complex inter-relationship is not well understood. Using instead a simple model that relies solely on the LLC and assumes a stable latency did not prevent our scheduler from performing successfully. Nevertheless, there is no limitation in CAMP that prevents the use of more accurate *SF* models. For instance, in [24], the authors use a similar approach for estimating *external stalls* (i.e., any stall due to resources external to the core), but their *SF* model also uses additional performance counters to account for *internal stalls* caused by branch mispredictions and the contention of other internal resources. We could extend CAMP with this additional metric, but we found it does not provide higher accuracy on our emulated AMP system since internal stalls are the same on both core types (they have the same microarchitecture).

In CAMP, LLC miss rates are measured for each thread continuously, and the values are sampled every 20 timer ticks (roughly 200ms on our experimental system). We keep a running average of the values observed at different periods and we discard the first values collected immediately after the thread starts or after it is migrated to another core in order to correct for cold-start effects causing the miss rate to spike intermittently after migration. We also use a carefully crafted mechanism to filter out transitions between different program phases. Updating *SF* estimations during abrupt phase changes may trigger premature changes in the *UF* and, as a result, unnecessary migrations, which may cause substantial performance overhead. Instead, *SF* estimations are updated exclusively once a thread enters a phase of stable behavior. To detect those stable phases, we used a light-weight mechanism based on a `phase_transition_threshold` parameter (12% in our experimental platform). When the running average is recorded, it is compared with the previous average measured over the previous interval. If the two differ by more

than the transition threshold, a phase transition is indicated. Two or more sampling intervals containing no indicated phase transition signal a stable phase.

On processors with shared caches, the thread's miss rate may vary due to the sharing of the cache with other threads, in addition to reasons related to internal program structure. For example, the miss rate may decrease because of co-operative data sharing or increase because of cache contention. However, we observed that the *quality* of the miss rate does not change significantly regardless if the thread shares a cache or runs solo; i.e., if the thread's miss rate is low relative to other threads when it runs solo, its value relative to other threads will stay low when it shares the cache even though it may increase by tens or hundreds of percent relative to its solo value. Similarly, if the thread's miss rate is high it will stay high relative to other threads, regardless if there is sharing.

We define three categories for the speedup factors and each category is labeled by a "representative" *SF* of that category. The representative *SF* is machine specific and was set empirically. The thresholds delimiting the categories were also chosen experimentally. After estimating a thread's *SF* we determine what category it fits in and assign it the *SF* equal to the label value corresponding to that category. In Section 6.3, we compare observed and estimated ratio and show that our model has enough accuracy to effectively guide scheduling decisions.

When computing the utility factor for a thread, we do not average the *SF*s of all threads in this application, but we use the *SF* of the thread in question. Averaging *SF* values would require cross-thread communication, which could damage the scalability of the scheduler. Using the current thread's *SF* is a good approximation of averaging for the following reason. First of all, in most applications we examined (see more about our selected benchmarks in Section 6.3) all threads do the same type of work, so their *SF* values would be the same. In applications where threads do different work, the most frequently occurring *SF* values will dominate and ultimately determine where most application is scheduled.

Finally, we describe an optimization related to the computation of the utility factor. The overhead of measuring the LLC miss rates is negligible at the sampling rate we use. However, we found during early development stages that computing the *UF* and updating the associated data structures at every sampling period may introduce some overheads. Fortunately, these can be substantially removed by applying certain optimizations. For instance, if we determine that a thread of a highly threaded application could never achieve a MEDIUM or HIGH utility factor even if it had the highest *SF* possible (i.e., the speed ratio between the fast and the slow cores), we do not recalculate the *SF* for the threads in this application unless the number of threads decreases, effectively removing the associated overheads.

6.2.2. The other schedulers

There are three other schedulers with which we compare the CAMP algorithm: Parallelism-Aware (PA), which delivers TLP specialization only; SF-Driven (SFD),

which delivers efficiency specialization only; and round-robin (RR), which equally shares fast and slow cores among all threads. We implemented all these algorithms in OpenSolaris. We do not compare with the default scheduler present in our experimental operating system, because the performance with this scheduler exhibited a high variance making the comparison difficult. We observed that this variance is especially high when running multi-application workloads consisted of single-threaded applications only. Nevertheless, RR's performance is comparable to or better than the default scheduler, so this is a good baseline.

The PA scheduler was described in detail in Chapter 5. For the comparison with CAMP, we reimplemented PA relying on the utility factor, which makes it possible for us to share a significant amount of code base between both schedulers' implementations. Because PA accounts only for TLP, the utility factor as such cannot be used to classify threads. For that reason the *SF* value in the *UF* formula (Equation 6.14) is replaced by a constant representing the upper bound of the achievable *SF* on a given system: the theoretical maximum for the instruction-per-second ratios between fast and slow cores. PA, like CAMP, boosts the fast-core priority of threads executing sequential phases of parallel applications by assigning them into SEQUENTIAL_BOOSTED class. However, since PA does not compute thread-specific speedup factors, it cannot distinguish between HIGH, MEDIUM and LOW utility threads. So unlike CAMP, which will place only high-utility threads in the SEQUENTIAL_BOOSTED class, PA will place all threads executing sequential phases in that class.

SFD, similarly to PA, uses the *UF* formula where the number of threads is always equal to one, since it does not account for the TLP of the application. The SFD scheduler estimates the *SF* using our method based on LLC miss rates. As the careful reader may observe, SFD is very similar to the HASS-D algorithm presented in Chapter 4. In fact, both HASS-D and SFD rely on the same technique for discovering which threads utilize complex cores most efficiently, without requiring cross-core migrations. The main difference between both scheduling algorithms is that SFD relies on utility classes to mitigate inaccuracies in the estimation of *SFs*, while HASS-D performs thread-to-core mappings based on "raw" *SFs*. In scenarios where SFs are estimated accurately for the entire workload, HASS-D may outperform SFD since it always guarantees that threads with the high SF run on fast cores, whereas SFD fair-shares fast cores among threads in a broader *high utility* class. When those predictions are not accurate[5], we observed that HASS-D's classless design may be subjected to incorrect mappings as well as to higher migration overheads stemming from the usage of raw estimated *SFs*. As a result, SFD is able to outperform HASS-D in most multi-application workloads evaluated in this chapter.

The RR algorithm shares fast and slow cores among threads using the same mechanism that CAMP uses to share fast cores among applications of the HIGH utility

[5] Let SF_A and ESF_A be the actual and predicted *SF* for an application A respectively. Given a workload W consisted of N applications, $A_1, A_2, \cdots, A_N$, we define that the estimation is *accurate* if for any pair of applications in the workload (A_i, A_j) where $SF_{A_i} \leq SF_{A_j}$, then $ESF_{A_i} \leq ESF_{A_j}$.

class. For the evaluation in this chapter, we used the HAFS algorithm, our real-world implementation of such a RR policy described in Section 4.2.4.

6.2.3. Topology-aware design

An important challenge in implementing any asymmetry-aware scheduler is to avoid the overhead associated with migrating threads across cores. Any asymmetry-aware scheduler relies on cross-core migrations to deliver the benefits of its policy. For example, CAMP must migrate a high utility thread from a slow core to a fast core if it detects that the thread is executing a sequential phase. Unfortunately, migrations can be quite expensive, especially if the source and target cores are in different *memory domains* of the memory hierarchy[6]. On NUMA architectures, remote memory accesses further aggravate this issue and migration cost can be even higher.

However, any attempt to reduce the number of migrations may backfire by decreasing the overall benefits of asymmetric policies. Apart from making the scheduler aware of applications' sensitivities to cross-memory-domain migrations, it is also worth considering ways to make migrations less expensive. In particular, if AMP systems are designed such that there is a fast core in each memory hierarchy domain (i.e., per each group of slow cores sharing a cache), migration overhead might be mitigated. As a matter of fact, in Section 5.2.4 we showed that the overhead of migrations becomes negligible with such *migration-friendly* designs, as long as the schedulers minimize cross-domain migrations. Based on these insights, our implementations of all the investigated schedulers have been carefully crafted to avoid cross-domain migrations when possible (i.e., all the schedulers are *topology aware*).

6.3. Experiments

The evaluation of the CAMP algorithm was performed on the AMD-16 and the Intel-8 platforms, both presented in Section 3.3. AMD-16 consists of sixteen cores organized into four quad-core AMD "Barcelona" CPUs. Intel-8 includes two Intel Xeon quad-core CPUs (8 cores).

Our asymmetric configurations consist of two core types: "fast" and "slow". The frequency of fast and slow cores was set to the maximum and minimum frequency (DVFS) levels, respectively. In particular, on Intel-8's asymmetric configurations, fast cores operate at 3.0 GHz, while slow cores run at 2.0 GHz (fast cores are 1.5 times faster than slow cores). On the AMD platform, in contrast, higher performance differences between core types are obtained since fast cores operate at twice the frequency of slow cores (at 2.3 GHz and 1.15 GHz, respectively). Notably, the AMD-16 platform supports core-level DVFS, so we are able to vary the frequency

[6] A memory hierarchy domain in this context is defined as a group of cores sharing a last-level cache.

for each core independently. On the Intel-8 platform, by contrast, cores in the same physical package (sharing an L2 cache) are within the same power domain, so they must operate at the same voltage/frequency level.

In our experiments we used four AMP configurations: (1) 1FC-12SC – one fast core and 12 slow cores, the fast core is on its own chip and the other cores on that chip are disabled; (2) 4FC-12SC – four fast cores and 12 slow cores, and (3) 2FC-2SC – two fast cores, two slow cores, none of them sharing a last-level cache with one another. Note that the first two configurations were emulated on AMD-16, while the third one was replicated on both Intel-8 and AMD-16.

We experimented with applications from the SPEC OMP 2001, the SPEC CPU 2006, and the Minebench suites, as well as `BLAST` – a bioinformatics benchmark – and `FFTW` – a scientific benchmark performing the fast Fourier transform. In all workloads (multi-application), we ensure that all applications are started simultaneously and when an application terminates it is restarted repeatedly until the longest application in the set completes three times. The observed standard deviation was negligible in most cases (so it is not reported) and where it was large we restarted the experiments for as many times as needed to guarantee that the deviation reached a low threshold. The average completion time for all the executions of a benchmark under a particular asymmetry-aware scheduler is compared to that under RR, and wall clock speedup is reported.

In all experiments, the total number of threads (sum of the number of threads of all applications) was set to match the number of cores in the experimental system, since this is how runtime systems typically configure the number of threads for CPU-bound workloads that we considered [26].

Our evaluation section is divided into three parts. In Section 6.3.1, we evaluate the accuracy of our method for estimating the speedup factor. In Section 6.3.2, we describe the workloads that we tested and briefly discuss results for single-threaded applications. In Section 6.3.3, we present the aggregate results for all workloads with all schedulers, and analyze the multi-threaded workloads in more detail.

6.3.1. Accuracy of SF estimation

In this subsection, we compare the estimated *SF* to the actual *SF* for all applications in SPEC CPU2006. The actual *SF* is measured by running the application on the slow core, then on the fast core, and computing the speedup. The estimated *SF* is obtained from the average LLC measured throughout the entire run of the application. Figure 6.3 shows the measured and the estimated ratios on our AMD and Intel systems respectively. Measured speedup ratios obtained in the environment where threads are periodically migrated between different cores are also measured – these data better reflect the realistic conditions under which the *SF* must be obtained

As Figure 6.3 shows, the estimates are accurate for CPU-intensive applications on both platforms (on the right side of the chart), but less accurate for mildly

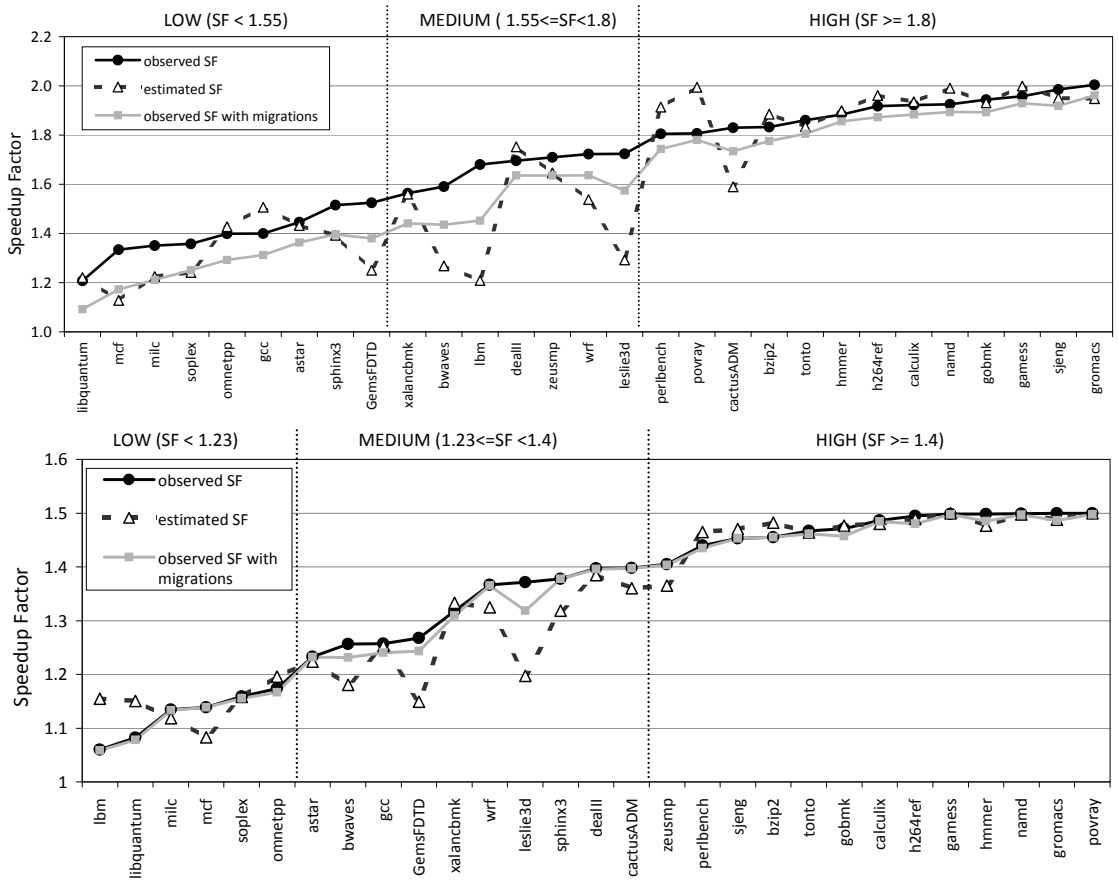


Figure 6.3: Observed and predicted speedup factors for all benchmarks of SPEC CPU2006 benchmarks on an AMD Opteron (top) and an Intel Xeon (bottom) platforms.

memory-intensive applications (on the center of the chart). As we explained earlier, inaccuracies occur as a result of the simplifying assumptions made in our model.

We observed that model inaccuracies are mitigated when it is used in the scheduler, because the scheduler categorizes applications into coarse Speedup Factor classes rather than relying solely on *SF* estimates. On both platforms, the thresholds that define these classes are set empirically to $\frac{1}{2}$ and $\frac{4}{5}$ of the maximum *SF* attainable, as shown in Figure 6.3.

6.3.2. Workloads

We experimented with two sets of workloads: those consisting of single-threaded applications, typically targeted by algorithms like SFD, and those including multi-threaded applications, typically targeted by algorithms like PA.

Single-threaded applications

To evaluate our scheduling algorithms for different types of applications and workloads, we selected eleven applications from the SPEC CPU 2006 suite and con-

Table 6.2: Multi-application workloads consisted of single-threaded applications

Categories	**Benchmarks**
4CI	`gamess`, `perlbench`, `povray`, `gromacs`
3CI-1MI	`sjeng`, `gamess`, `gromacs`, `soplex`,
2CI-2MI_A	`perlbench`, `povray`, `soplex`, `mcf`
2CI-2MI_B	`gromacs`, `sjeng`, `milc`, `soplex`
1CI-3MI_A	`gamess`, `milc`, `soplex`, `mcf`
1CI-3MI_B	`gromacs`, `milc`, `soplex`, `GemsFDTD`
4MI	`GemsFDTD`, `milc`, `soplex`, `mcf`
Phased1	`astar`, `astar`, `milc`, `leslie3d`
Phased2	`sjeng`, `astar`, `milc`, `leslie3d`
Phased3	`astar`, `astar`, `GemsFDTD`, `GEMSFDTD`

structed ten workloads containing representative pairs. In selecting applications, we tried to cover a wide variety of behaviors. Some benchmarks are either memory intensive (such as `mcf` and `milc`) or CPU intensive (such as `gromacs` and `sjeng`), whereas others exhibit different phases across their execution (`astar` is a memory-intensive application that also exhibits some CPU-intensive phases).

The ten workloads shown in Table 6.2 cover a rich set of scenarios. 4CI and 4MI are homogeneous workloads that combine applications of the same class (either CPU-intensive or memory-intensive applications) and xCI-yMI are heterogeneous workloads that mix memory-intensive and CPU-intensive applications. The categories in the left column are listed in the same order as the corresponding benchmarks, so for example in the 1CI-3MI category `gromacs` is the CPU-intensive (CI) application and `milc`, `soplex` and `mcf` are the memory-intensive (MI) applications. The last three workloads labeled as "Phased" include applications that do not fall into a clean class since they exhibit different phases.

The results for these workloads running under PA, CAMP and SFD are shown in Figure 6.4. To complement our assessment on the effectiveness of SF predictions, we also provide a comparison with a "Best Static" assignment, which ensures applications with the highest overall ratios to run on fast cores. As expected, PA behaves like RR since it is unaware of the efficiency of individual threads and, as a result, fair-shares fast cores among them. (Recall that PA assigns all single-threaded applications to the HIGH utility class.) CAMP and SFD perform similarly, since $UF{=}SF$ for single-threaded applications. Overall, we observed that these algorithms effectively distinguish between CPU-intensive and memory-intensive code and perform thread-to-core mappings closer to the "Best Static" in the absence of phase changes (on the Intel platform, SFD and CAMP behave better due to the higher accuracy of the SF estimations). For applications that exhibit different phases across their execution, "Best Static" does not guarantee optimal mappings.

Single-threaded and multi-threaded applications

We categorized applications into three groups with respect to their parallelism: highly parallel applications (HP), partially sequential (PS) applications (parallel

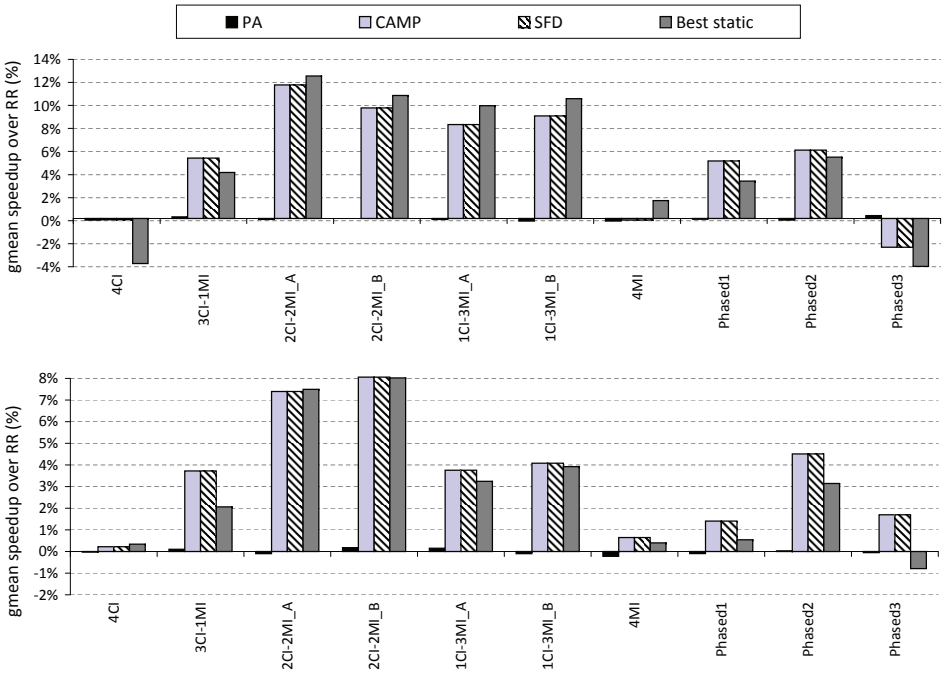


Figure 6.4: Speedup of PA, SFD, CAMP and Best Static schedulers when running single-threaded workloads on the 2FC-2SC AMD (top) and Intel (bottom) platforms.

applications with a sequential phase of over 25% of execution time), and single-threaded applications (ST). In order to cater to application memory intensity, we divided, in turn, the three aforementioned groups into memory-intensive (MI) and CPU-intensive classes (CI), resulting in six application classes: HPCI and HPMI classes for highly parallel applications, CPU intensive and memory intensive, respectively; PSCI and PSMI classes for partially sequential applications, and STCI and STMI classes for single-threaded applications.

We constructed nine workloads consisting of representative pairs of benchmarks across the previous categories mentioned above as shown in Table 6.3. As in Table 6.2, the categories in the left column are listed in the same order as the corresponding benchmarks. For example, in the STCI-PSMI category `gamess` is the single-threaded CPU-intensive (STCI) application and `FFTW` is the partially sequential memory-intensive (PSMI) application. The numbers in parentheses next to the application class indicate the number of threads chosen for that application: the first number for the 1FC-12SC configuration and the second number for the 4FC-12SC configuration.

At a first glance, it can be observed from the workloads that not all possible pairs of classes are actually covered. For the sake of analyzing benchmark pairings that expose diversity in instruction-level and thread-level parallelism, we did not pick pairs consisting of co-runners of the same class. Note also that highly parallel memory-intensive benchmarks have been deliberately discarded from these workloads. In preliminary experiments we observed that for benchmark pairings with a highly parallel application (either HPCI or HPMI), schedulers that rely on the number of threads when making scheduling decisions (CAMP and PA) mapped all threads of

Table 6.3: Multi-application workloads with both single-threaded and multi-threaded applications

Categories	Benchmarks
STCI-PSMI	gamess, FFTW (12,15)
STCI-PSCI	gamess, BLAST (12,15)
STCI-HP	gamess, wupwise_m (12,15)
STMI-PSMI	mcf, FFTW (12,15)
STMI-PSCI	mcf, BLAST (12,15)
STMI-HP	mcf, wupwise_m (12,15)
PSMI-PSCI	FFTW (6,8), BLAST (7,8)
PSMI-HP	FFTW (6,8), wupwise_m (7,8)
PSCI-HP	BLAST (6,8), wupwise_m (7,8)

Table 6.4: Additional multi-application workloads with both single-threaded and multi-threaded applications

Categories	Benchmarks
4CI-1PSMI-1HP	gobmk, h264ref, gamess, povray, FFTW(6), wupwise_m(6)
4CI-4MI-1HP	games, gobmk, h264ref, gromacs, milc, mcf, soplex, libquantum, equake_m(8)
4CI-4MI-1PSMI	calculix, hmmer, gamess, sjeng, milc, mcf, soplex, libquantum, FFTW(8)
3CI-1MI-1PSCI	gamess, gobmk, hmmer, soplex, semphy(12)

the HP application on slow cores. The actual reason behind this behavior is that a high number of active threads (this happens most of the time for HP applications) dominates the value of the utility factor and, as a result, CAMP and PA schedulers always assign a LOW utility class for all threads, regardless of their memory intensity (*SF*). For that reason, we only included wupwise_m as a representative HP application (a CPU-intensive parallel benchmark from SPEC OpenMP 2001).

For the sake of completeness, we have also studied additional multi-application workloads that combine parallel- and single- threaded applications, but exhibit a wider variety of memory intensity than those in Table 6.3, which focus on exploring the impact of thread-level parallelism. Table 6.4 shows this additional set. Notably, as opposed to the benchmarks pairings shown in Table 6.3, the additional workload set includes HP memory-intensive (HPMI) applications, such as equake_m.

Both OpenMP and POSIX threaded applications use adaptive synchronization modes, as such, large sequential phases are exposed to the operating system in both cases. Nevertheless, applications implemented using POSIX threads (BLAST, FFTW) spin for shorter periods of time before blocking (these are the default parameters used in OpenSolaris).

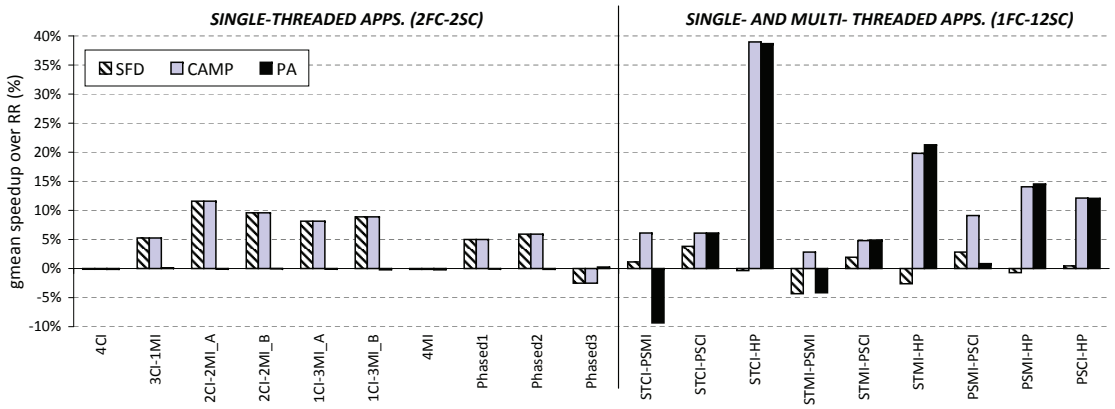


Figure 6.5: Gmean speedup of SFD, PA and CAMP schedulers when running single threaded and multi-threaded workloads on the AMD platform.

6.3.3. Aggregate results and detailed analysis of multi-threaded workloads

Figure 6.5 shows the geometric mean of the speedups achieved by the three asymmetry-aware schedulers (SFD, PA and CAMP) normalized to RR, when running on the AMD platform. Only CAMP is able to deliver performance gains across the wide variety of workloads analyzed in our study, which is the major contribution of this research.

Before discussing in detail per-application results, it is worth analyzing the behavior of the partially sequential applications included in the workloads: `BLAST` (PSCI) and `FFTW` (PSMI). As opposed to other parallel applications that create all threads at the beginning of the execution, both `BLAST` and `FFTW` exhibit several distinct parallel phases where threads are destroyed at the end of a phase and new threads are created at the beginning of the subsequent one. When scheduled by algorithms relying on on-line *SF* monitoring (CAMP and SFD), new spawned threads will have to go through the initial `warm_up` period until they are eligible to be scheduled on fast cores. This means that frequent thread creation and destruction might imply that threads will be running on slow cores more often. Moreover, it is worth noting that both `FFTW` and `BLAST` have significant serial bottlenecks (over 40% of total execution time) and CPU-intensive parallel phases.

Serial phases in `FFTW` (memory intensive) comprise roughly 80% of the total execution time, so we can globally categorize this application as memory intensive. By analyzing per-thread behavior over time using performance monitoring counters, we found that `FFTW`'s serial phases are, in turn, divided into a very short CPU-intensive phase (at the beginning) and a long memory-intensive phase. According to the boosting feature incorporated into CAMP, the thread executing a sequential phase is initially assigned to the SEQUENTIAL_PART class, since the thread starts exhibiting a CPU-intensive behavior. Later on, when the serial thread enters the memory-intensive phase, CAMP downgrades it into the MEDIUM class.

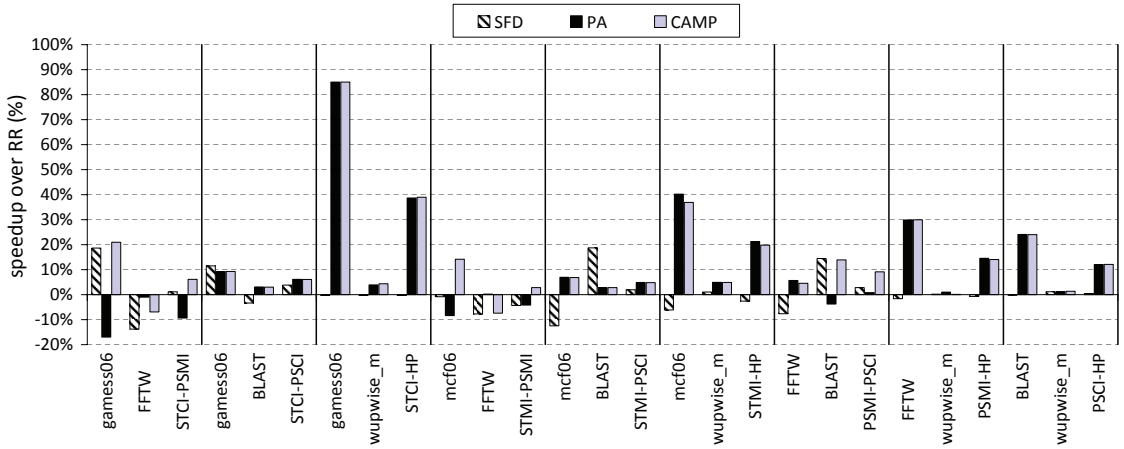


Figure 6.6: Speedup of asymmetry-aware schedulers on 1FC-12SC.

PA, as well as CAMP supports explicit *boosting* of the priority for running on a fast core for a thread executing a sequential phase of the application. After exploring the effect of varying the customizable parameter `amp_boost_ticks`, we set it to one hundred time slices (1 second), which ensures the acceleration of sequential phases without monopolizing the fast core.

Figures 6.6 and 6.7 show the results for the 1FC-12SC and 4FC-12SC configurations respectively. There is a speedup bar for each application in the workload as well as the mean speedup for the workload as a whole labeled with the name of the workload from Table 6.3. The first thing to highlight is that RR behaves well when there are just a few threads running in the system since all of them will get a significant "share" of fast-core cycles. Workloads with few threads include those with two PS applications (recall that our PS applications have large phases where only a single thread is active) as well as with one ST and one PS application. Now, we analyze each workload separately for the 1FC-12SC configuration:

- (STCI-PSMB) PA boosts the large sequential phase of `FFTW` (memory intensive) at the expense of scheduling the CPU-intensive sequential application (`gamess`) on slow cores. RR, in contrast, shares the fast cores between the sequential phase of `FFTW` and `gamess`, behaving better than PA as a result. CAMP only schedules `FFTW` on the fast core during the initial CPU-intensive portion of its sequential phase, leaving the fast core available for `gamess` most of the time. Since `gamess` is CPU intensive, this is the right way to schedule, and so CAMP beats both RR and PA. SFD primarily runs `gamess` on the fast core, failing to accelerate the sequential phase of `FFTW`.

- (STCI-PSCI) PA and CAMP behave similarly here, because `BLAST`'s sequential phase is also CPU intensive, so both PA and CAMP schedule it on a fast core. In contrast, RR still schedules `BLAST` threads on the fast core when it is executing a parallel phase (many active threads), reducing `gamess`'s share of fast-core time. Surprisingly, SFD schedules `gamess` on the fast cores more often than RR does. The reason behind this behavior is that, as stated previously, `BLAST` creates and destroys threads several times and as a result new

spawned threads are not eligible to be scheduled on fast cores until expire their warm-up period. During these periods, `gamess` is the only CPU-intensive application eligible to run on fast cores.

- (STCI-HP) In this scenario, many CPU-intensive threads are active throughout the execution. RR and SFD perform similarly as a result of fair-sharing the fast core among all threads. On the other hand, PA and CAMP schedule the single-threaded application in the HIGH utility class (`gamess`) on the fast core all the time, leaving slow cores for `wupwise_m`'s LOW utility threads. For this reason, CAMP and PA perform significantly better than RR.

- (STMI-PSMI) CAMP does not have many opportunities to improve performance relative to RR here. Both `mcf` and `FFTW` are primarily memory intensive, and RR shares the fast core among them. CAMP beats RR by a small amount, only because it schedules `FFTW` on the fast core during the CPU-intensive portion of its sequential phase. PA primarily schedules `FFTW` on the fast core due to its large sequential phase, which PA is configured to maximally accelerate. As a result, `mcf`, an application with a slightly greater *SF* than the memory-intensive part of `FFTW`, runs mostly on the slow core.

- (STMI-PSCI) CAMP and PA, which perform similarly here, schedule the single-threaded `mcf` on the fast core as long as `BLAST` is running a parallel phase. When `BLAST` enters a sequential (CPU-intensive) phase, its active thread is executed on the fast core, pushing `mcf` to the slow core. SFD, however, runs the memory-intensive `mcf` on a slow core while running `BLAST`'s threads on both fast and slow cores, since those threads have a CPU-intensive nature and thus a high speedup factor.

- (STMI-HP) This workload is similar to STCI-HP, since most threads are active for the duration of the experiment. PA schedules the single-threaded application on the fast core and so does CAMP; therefore, they perform similarly. In contrast, SFD schedules `mcf` on the slow core, since this is the only memory-intensive thread in the workload.

- (PSMI-PSCI) The performance differences between PA and CAMP in this scenario are dominated by the fact that `FFTW`'s sequential phases are on average much longer than `BLAST`'s. Under the PA scheduler, the first thread executing an application's sequential phase is placed on the fast core and will not be migrated from it until `amp_boost_ticks` expire or the thread blocks. The long sequential phases of `FFTW` leads to monopolizing the fast core and `BLAST`'s sequential phases have little chance to run there, since PA does not share the fast cores equally among threads in the SEQUENTIAL_BOOSTED class. As a result, PA does to cater to the greater efficiency of `BLAST` in using fast cores, and resorts instead to running `FFTW`'s memory-intensive sequential phases on fast cores. CAMP, however, is able to detect `FFTW`'s memory-intensive sequential phases, successfully downgrading the thread executing it into the MEDIUM class.

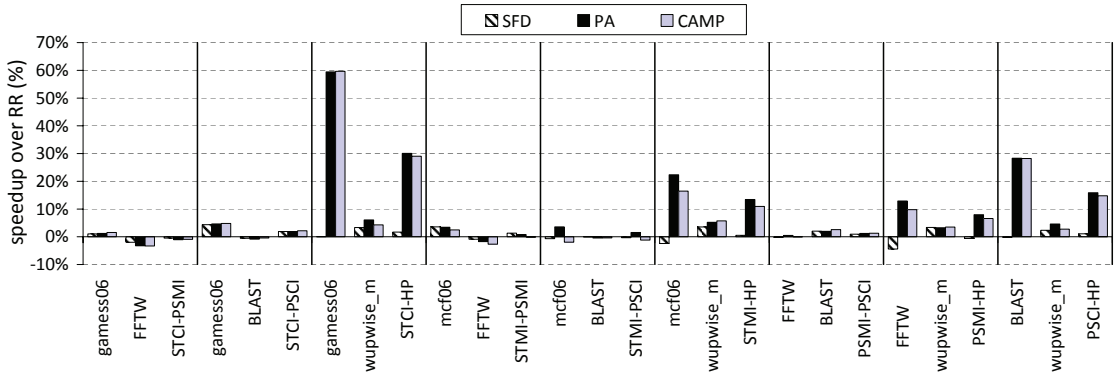


Figure 6.7: Speedup of asymmetry-aware schedulers on 4FC-12SC.

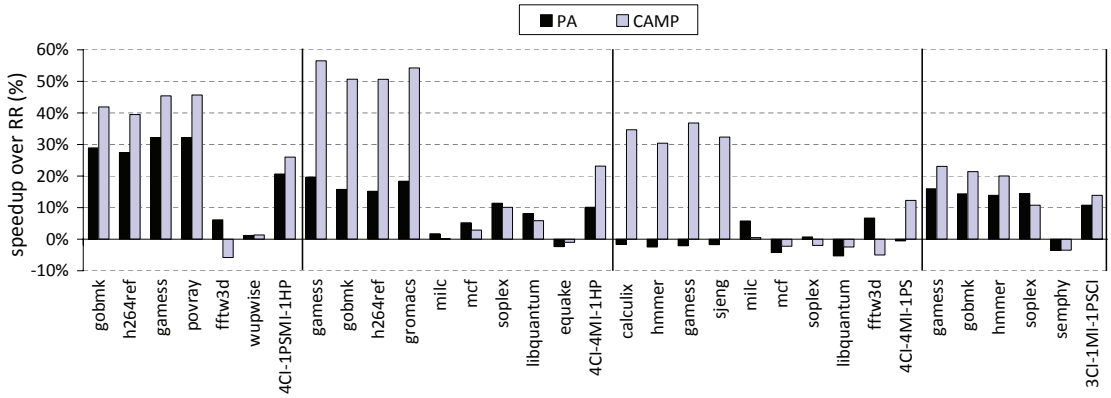


Figure 6.8: Speedup of the PA and CAMP schedulers for additional workloads on 4FC-12SC.

- (PSMI-HP) Sequential phases of the PS applications are effectively accelerated by PA and CAMP on the fast core. SFD, on the other hand, is not able to deliver any performance gains, because it schedules the memory-intensive sequential phases of `FFTW` on slow cores, running on the fast core CPU-intensive threads of parallel (wupwise_m), which gains little speedup when only one of its threads is accelerated.
- (PSCI-HP) As in the PSMB-HP workload, the thread executing sequential phases of the PS application is migrated to the fast core by PA and CAMP. SFD, in contrast, shares the fast cores among all threads, since they are CPU intensive and, as a result, it behaves as RR.

In Figure 6.6, CAMP and PA performed comparably in most cases, because they both considered TLP. CAMP only outperforms PA on 1FC-12SC when a single-threaded application and a memory-intensive serial thread compete for a fast core. However, on 4FC-12SC for the workloads in Table 6.3 (same benchmarks, different number of threads), PA and CAMP always perform similarly since both schedulers have enough fast cores to effectively accelerate single-threaded applications as well as serial threads (Figure 6.7).

Therefore, there still remains a question: if considering the speedup factor *in addition to* TLP is important for multi-threaded workloads, and in what cases it can

bring significant performance improvement over an algorithm that relies on TLP only. Results in Figure 6.8 answer this final question showing additional workloads with a wider diversity in memory intensity. In these cases, CAMP does deliver greater performance gains over PA (up to 13%), which demonstrates that considering the speedup factor in addition to TLP brings higher performance improvements.

6.4. Related work

Several asymmetry-aware schedulers were proposed in previous work. Most of them delivered either ILP specialization (see Chapter 4) or TLP specialization (see Chapter 5), but not both. As a result, existing AMP schedulers addressed parts of the problem, but did not provide a comprehensive solution: one that would address a wide range of workloads as opposed to targeting a selected workload type. The CAMP scheduler described in this chapter enables us to close this gap.

The evaluation of the CAMP scheduler was carried out on emulated asymmetric configurations, which were built by downscaling the frequency of some cores cores in symmetric systems. Although this is not the most accurate way to approximate future AMP platforms, in which the cores of different types are also likely to have different pipeline architectures (e.g., simple in-order vs. complex out-of-order), this methodology permitted us to run our experiments on a real system[7], as opposed to a simulator. Nevertheless, work by other researchers (carried out concurrently with the design and evaluation of CAMP) suggests that conclusions made in our work would apply to asymmetric systems with more profound differences between core microarchitectures. We are referring to the work by Koufaty, Reddy and Hahn from Intel [24], where the authors used proprietary tools to emulate an asymmetric system where the cores differed in the number of micro-ops that could be retired per cycle. The authors assumed cores of two types: a *big* core capable of retiring up to four micro-ops per cycle, and a *small* core capable of retiring at most one micro-op per cycle. For single threaded applications from the SPEC CPU 2006 suite, they found that the relative speedup on the big core relative to a small core highly correlates with the amount of *external stalls* generated by the application, which are in turn approximated by memory reads and requests for cache line ownership. The CAMP and SFD schedulers presented in this chapter approximate relative speedups using last-level cache miss rates, which include the metrics used in the aforementioned work and would have a high correlation with them. In conclusion, this suggests that the findings of our work could have direct application to systems with more significant differences between the cores than in our experimental system.

While most asymmetry-aware schedulers were concerned with two types of specialization (related to the workload's ILP and parallelism), a few researchers looked at less conventional types of specialization[8]. Mogul et al. proposed using slow cores in

[7] As a result, we were able to do a much more extensive analysis than what would have been possible on a simulator.

[8] More information on this topic can be found in Section 2.1.3.

asymmetric systems for executing system calls [13], since system calls are dominated by code that uses fast and powerful cores inefficiently. They modified an operating system to switch the execution to a less powerful core when a thread enters the system call. The performance improvements achieved by Mogul's scheduler, however, were not very large due to the overhead associated with thread migrations aimed to execute system calls on slow cores. Nevertheless, our implementation of CAMP partially exploits this kind of core specialization since all Solaris's kernel threads are forced to run on slow cores. Also inspired by this idea, Kumar and Fedorova proposed binding the controlling domain *dom*0 of a virtual machine monitor Xen to a slow core [82]. They also observed that workloads dominated by activity in *dom*0 are less affected by variations in the speed of the core than other workloads.

The asymmetry-aware schedulers discussed up to this point were targeted at optimizing overall system efficiency, as opposed to performance of individual threads. Other researchers addressed fairness and attempted to mitigate any negative effects that asymmetry may bring, such as unfair speedup received by different threads and unstable execution times.

One of the first schedulers addressing this problem was described by Li et al. [17]. As described in Section 2.3.2, Li's scheduler allowed fast cores to receive a higher load than slow cores. This was done to account for the fact that fast cores perform more work per unit of time. Li's scheduler did not take into account, however, that different threads experience different rates of speedup when running on a fast core relative to slow cores. As a result, under certain workloads it was possible to run into a situation where threads assigned to fast cores make slower progress than threads assigned to slow cores. Li's scheduler implicitly addressed fairness: threads running on fast cores would accomplish more work per unit of time than threads running on slow cores, but they would receive a smaller share of CPU time because fast cores were given a higher load. On the other hand, if the number of threads in the workload did not exceed the number of cores, Li's algorithm would not deliver fairness. To address fairness in this scenario, the scheduler needs to periodically rotate threads among fast and slow cores in a round-robin fashion. Such asymmetry-aware round-robin scheduler was described by Fedorova et al. [81]. Balakrishnan et al. described a simple asymmetry-aware scheduler that ensured that fast cores do not go idle before slow cores [14]. Despite its simplicity the scheduler significantly improved performance stability of realistic workloads.

The schedulers described so far targeted explicitly asymmetric systems – systems where asymmetry was introduced by design or emulated using the techniques described in Section 3.1. Another category of schedulers addressed systems where asymmetry was introduced "by accident", for example as a result of variation in the fabrication process. These variations cause cores to run at varying frequencies and consume varying amounts of power [83]. Therefore, the assumption that there are only two types of cores no longer applies. Other than that, the problem is similar to that on explicitly asymmetric systems: how to match threads to cores so as to minimize some global function, for example delay or energy/delay product.

Three algorithms addressing this problem were described by Winter and Albonesi [84].

These algorithms relied on sampling performance of threads on each (or most) core types. The first algorithm, the Hungarian algorithm, was used to solve the weighed bipartite matching problem and relied on an $N \times N$ matrix, where N is the number of cores and threads. An entry (i, j) in the matrix corresponded to the cost of running a process i on core j. In this case, cost was represented by ED^2 (energy/delay squared). Rows and columns of the matrix were manipulated according to the algorithm to find an assignment that minimized total ED^2. The second algorithm considered by Winter and Albonesi relied on global iterative search. The scheduler tried various randomly chosen thread assignments and then used the one with the lowest total cost. This strategy was similar to one of the algorithms proposed by Kumar et al. [3] for explicitly asymmetric systems. The final algorithm explored by Winter and Albonesi used local search. Local search is different from global search in that it probes assignments that are "close" to the initial one in the search space – for example an assignment that can be derived from the initial one with a small number of swaps in the assignment of threads. The Hungarian algorithm turned out to perform the best, but its complexity is $O(N^3)$ in the number of cores. This made it an unlikely candidate for systems with hundreds or thousands of cores. Local search, on the other hand, had a lower complexity ($O(N)$), but performed similarly to the Hungarian algorithm.

Algorithms designed by Winter and Albonesi required sampling of threads on different core types, and in particular the Hungarian algorithm required that each thread is sampled at each frequency. As we showed in Chapter 4, this requirement may cause problems, such as incorrect speedup estimates and load imbalance. Furthermore, on systems where asymmetry is caused by process variation and the number of different core types may be substantially larger than two, sampling complexity further increases. An algorithm proposed by Ghiasi et al. overcame this shortcoming [15]. Their algorithm also targeted systems where cores have the same microarchitecture, but run at varying frequencies, but it did not require sampling of threads' runtimes on different cores to construct a good assignment of threads to cores. Instead, Ghiasi used a performance model to predict how a thread's IPS (instructions per cycle) is affected by a change of frequency. This performance model, similar to the one described in Section 6.2.1, was based on the number of off-core request issued by the thread. Ghiasi's algorithm was implemented and evaluated on a real system, and so it was built with practical considerations in mind: the algorithm was structured in a way that did not require examination of all possible mappings of threads to cores – a task that could be prohibitively expensive on a system with a large number of frequencies.

6.5. Conclusions

In this chapter, we have presented a comprehensive scheduling algorithm for asymmetric multicore processors. Although the advantages of exploiting ILP and TLP parallelism on AMPs were well understood before, no one had addressed the design of the corresponding *unified* support in the operating system and evaluated its

benefits and drawbacks. Previous asymmetry-aware schedulers employed only one type of specialization (either efficiency of TLP), but not both. As a result, they were effective only for limited workload scenarios.

Through our evaluation of a real OS implementation on real hardware, we determined that the CAMP scheduler can be effective for a wide variety of applications without requiring their modification. SFD is unable to deliver performance comparable to CAMP for workloads that include multi-threaded applications, while PA is unable to compete with CAMP when applications exhibit a wide variety of memory-intensity. Key elements for the success of CAMP are the Utility Factor (UF) and its light-weight mechanism for estimating per-thread fast-to-slow relative speedups.

Chapter 7

Conclusions

7.1. Thesis conclusions

Recent research has highlighted the potential benefits of single-ISA asymmetric multicore processors (AMPs) over cost-equivalent symmetric ones (SMPs), and it is likely that future processors will integrate cores that have the same instruction set architecture but offer different performance and power characteristics [1, 5]. Asymmetric designs are very appealing because they can achieve high single-threaded performance and at the same time, deliver high performance thread-level parallelism with lower energy costs.

A large body of work has demonstrated that this potential of AMPs is realizable via *core specialization* techniques, namely, the utilization of each type of core for the sort of computation where it delivers the best performance/energy trade-off. Unfortunately, specialization will not be delivered by the hardware, but it is up to the system software to deliver the potential of asymmetric systems to unmodified applications.

While the design of AMP processors has been extensively investigated and their potential benefits have been illustrated via theoretical or simulation-based analysis, the study of real-world operating system support for these upcoming architectures has not been addressed comprehensively to date. The main goal of this thesis was to fill this gap by investigating how and to what extent core specialization techniques can be effected via OS scheduling. To that end, we proposed three asymmetry-aware schedulers (HASS, PA and CAMP), whose implementation in an actual operating system was extensively evaluated on emulated asymmetric hardware and compared to previously proposed scheduling schemes.

Our Heterogeneity-Aware Signature-Supported (HASS) scheduling algorithm leverages knowledge on the microarchitectural diversity of the workload to maximize performance on AMPs. On the high level, HASS identifies threads in the workload that derive a higher benefit (speedup factor) from running on complex cores relative to simple ones, and maps them to complex cores; the remaining threads are relegated

to low-power cores. To make that possible, HASS relies on *architectural signatures*, which contain information about applications' runtime properties, such as their memory-access patterns or amount of instruction-level parallelism (ILP). We proposed two versions of HASS, static (HASS-S) and dynamic (HASS-D), which rely on offline- and online- generated signatures, respectively. The main insights emerging from our experimental evaluation are as follows:

- Benefits from asymmetry-aware scheduling algorithms are especially pronounced for workloads where there is a large disparity between applications' architectural properties and on systems with large differences in the speed among different types of cores. In particular, our experimental results reveal that HASS always outperforms an asymmetry-agnostic scheduler (up to 12.5% reduction in completion time) for workloads consisting of single-threaded applications exhibiting sufficient diversity.

- Contrary to our expectations, our implementation of HASS, both the static and the dynamic version, outperformed the previously proposed IPC-Driven algorithm, which relied on actual measured speedup factors as opposed to the estimated ones. We discovered that the IPC-Driven algorithm suffered from inaccuracies and overheads stemming from the need to measure performance on multiple core types.

- We found that the usage of offline collected architectural signatures rather than online ones incurs a lot less overhead at runtime, and this leads the static version to delivering greater performance gains. However, in the event signatures are not available (i.e., not embedded in the application binary) or these are not highly representative at runtime (e.g., when the application exhibits large and fairly distinct program phases or the signature greatly varies with the program input), online estimated signatures can be effectively used to fill this gap. Hence, HASS-D makes a more versatile scheduling solution for general-purpose asymmetric multicore systems.

In this thesis, we have also studied the capability of asymmetric multicore systems to mitigate scalability bottlenecks in parallel applications by accelerating their serial phases on fast cores. We carried out this study by means of the proposed Parallelism-Aware (PA) scheduler, the first OS-level scheduling algorithm exploiting this striking capability of AMPs. The obtained results led us to the following conclusions:

- Simple asymmetry-aware algorithms that keep fast cores as busy as possible in an AMP accelerate serial phases automatically for a trivial case of one parallel application with serial bottlenecks running alone in the system. Nevertheless, in multi-application scenarios or in the event that unused threads of parallel applications spin rather than blocking, these simple schedulers do not accomplish this task. In these scenarios, the PA scheduler outperforms simpler algorithms by up to 40%.

- By monitoring an application's runnable thread count, the PA scheduler can trivially detect its serial phases as long as its unused threads block at syncronization primitives. However, in the event unused threads busy-wait (or spin) during short periods of time, the OS cannot detect these phases simply by monitoring the runnable thread count. To address these scenarios, we designed PA Runtime Extensions (PA-RTX), a simple API enabling the user-level threading library (or runtime system) to notify the scheduler when a thread spins rather than doing useful work. Our experimental evaluation revealed that further performance gains were possible thanks to these extensions, which underscores the importance of the interaction between the runtime system and the OS when it comes to effectively detecting phases with limited parallelism in multithreaded software.

- Cross-core migrations are an essential mechanism in any asymmetry-aware scheduler. Unfortunately, migrations can be especially costly if the source core is in the different domain of the memory hierarchy than the target core (i.e., the two cores do not share a last level cache). In the quest of less expensive migrations, we found that migration overhead can be mitigated on systems where at least one fast core would be in the same memory-hierarchy domain with several slow cores (*migration-friendly* topology), and where the scheduler would avoid cross-domain migrations when possible (*topology-aware* design). In our opinion, this finding provides crucial insight for designers of future AMP systems.

Through experimental evaluation we demonstrated that both HASS and PA effectively maximize performance of AMPs for diverse workloads. These two schedulers are clear examples of the two broad categories of asymmetry-aware schedulers proposed to date: the former caters to the microarchitectural diversity of the workload and the latter exploits the diversity in thread-level parallelism. Because both kind of schedulers only account for either ILP or TLP of the applications (but not both), they are only capable of delivering significant benefits for limited workload scenarios.

This apparent limitation led us to proposing the CAMP scheduler. CAMP makes a comprehensive scheduling solution thanks to the *Utility Factor* (UF), a novel metric that accounts for both the ILP and TLP of the application and produces a single value that approximates how much the application as a whole will improve its performance if its threads are allowed to occupy all the fast cores available on that system. Our main findings can be summarized as follows:

- Schedulers catering to the microarchitectural diversity are unable to deliver performance comparable to CAMP for workloads that include multi-threaded applications. Conversely, algorithms exploiting the diversity in thread-level parallelism, like PA, are unable compete with CAMP in scenarios consisting of single-threaded applications only and whenever the workload exhibits sufficient memory intensity.

- An essential element for the success of CAMP is a new light-weight technique for discovering which threads utilize fast cores most efficiently.

Although the performance improvements achieved on our experimental systems were quite significant, we believe that on "real" asymmetric system (as opposed to emulated systems, like ours) they would be even greater. Future asymmetric systems are likely to have more drastic differences among the cores of different types (e.g., differences in the pipeline microarchitecture [1]), and we have shown that more heterogeneous hardware renders greater performance improvements from asymmetry-aware scheduling.

Furthermore, in future asymmetric multicore systems, memory access will likely remain as the major performance-limiting factor (as recently found in [24]). Therefore, we strongly believe that the number of off-core and off-chip requests (last-level-cache accesses and misses) can still be used by the thread scheduler, at least as a first approximation, to estimate the relative benefit between cores with different cache hierarchies and microarchitectures.

Our overarching conclusion is that in terms of potential for improving performance of software, AMP systems are a viable future alternative to symmetric systems as long as they are equipped with the right operating system support.

7.2. Future work

In this thesis, we focused on maximizing system-wide performance on AMPs via thread scheduling. Nevertheless, further factors and goals contribute to effective job scheduling on AMPs, among which we highlight the following:

- **Quality of Service (Qos) and priority enforcement:** The scheduling policies proposed in this thesis are inherently unfair in scenarios where some applications have a higher priority than others or in the event QoS guarantees are required. Nevertheless, priority-oriented policies for AMPs must still cater to the relative benefit that high-priority applications would derive from using fast cores –based on their amount of thread-level and instruction-level parallelism– in an attempt to provide better service for these applications while making efficient use of the AMP. For example, among high-priority compute-intensive applications, phases with low thread-level parallelism should be preferentially mapped to fast cores, since devoting the "scarce" fast cores of an AMP to running highly parallel phases yields modest performance gains at the expense of increasing power consumption significantly [4].
- **Power management:** In our experimental evaluation, we have assumed that a thread's fast-to-slow performance ratio remains constant within stable execution phases, since program-phase behavior does not depend upon the executing processor [53]. Unfortunately, the action of hardware- or software-

driven power-management mechanisms supported by modern processors may result in dynamic changes in some characteristics of the cores (such as the processor frequency) and, in turn, in variations of the actual performance ratio between fast and slow cores. As a result, the scheduler must react to these changes to properly update per-thread fast-to-slow speedups. Furthermore, power management and asymmetry-aware scheduling must be performed in a fully coordinated fashion to ensure effective utilization of the asymmetric platform under these circumstances. All in all, we believe that the study of performance asymmetric architectures coupled with dynamic changes in the performance of the different cores is an interesting avenue for future work.

- **Cache-conscious scheduling for AMPs:** Another interesting direction is to study the interactions between thread migrations and caching behavior. Numerous studies have shown that some applications are more sensitive to cross-memory-domain migrations than others due to the nature of their memory-access patterns [85, 86, 17], and if threads share cached data the problem becomes even more challenging [87]. Incorporating cache awareness into asymmetry-aware algorithms like CAMP would be a first step toward designing an all-encompassing scheduling algorithm for asymmetric multicore systems.

- **Estimating relative speedups on highly asymmetric architectures:** Most estimation models developed so far were evaluated on systems where cores have identical microarchitecture but differ in frequency (such as the ones explored in this thesis) or in retirement width [24]. What is missing, however, is a model that would work on asymmetric systems where cores differ more dramatically. We strongly believe that the first natural step to accomplish this is to devise robust methodologies that, regardless of where the performance differences among cores come from, enable us to identify those platform-specific performance metrics that best aid the thread scheduler in approximating relative speedups.

Apéndice A

Resumen en Español

En cumplimiento del artículo 4.3 de la normativa de desarrollo de los artículos 21 y 22 del R.D. 1393/2007 por el que se regulan los estudios universitarios oficiales de posgrado de la Universidad Complutense de Madrid, se presenta a continuación un resumen en español de la presente tesis que incluye introdución, objetivos, principales aportaciones y conclusiones del trabajo realizado.

A.1. Introducción

Las arquitecturas masivamente paralelas construidas mediante la integración de múltiples cores de consumo moderado son una de las apuestas más firmes de la industria y la academia para seguir mejorando las prestaciones de los computadores teniendo en cuenta las perspectivas tecnológicas actuales [6, 9, 10, 50]. Se estima que el número de cores por chip se incrementará en los próximos años de forma sostenida con cada nueva generación tecnológica, mientras que el rendimiento por core individual se mantendrá esencialmente plano.

La mayoría de los procesadores *multicore* actuales están constituidos por cores idénticos, incluyendo cores complejos desde el punto de vista de la microarquitectura –como la gama Xeon de Intel u Opteron de AMD–, o bien cores más simples de consumo reducido –como Sun Niagara [11] o Intel Atom–. Los procesadores del primer grupo incorporan sofisticadas características microarquitectónicas, como ejecución fuera de orden y superescalar, que contribuyen notablemente a incrementar el rendimiento mono-hilo. Lamentablemente, estudios previos advierten que los procesadores de este tipo no podrán integrar más allá de un número moderado de cores por problemas de consumo y disipación [3]. Los procesadores del segundo grupo prometen mejores prestaciones para aplicaciones con elevado paralelismo a nivel de hilo (TLP), pero ofrecen un rendimiento notablemente inferior para aplicaciones puramente secuenciales.

Como queda patente, la elección del tipo de sistema multicore *simétrico*, constituido por unos pocos cores complejos o bien por abundantes cores simples de bajo

consumo, determina el tipo de aplicación para el que el diseño escogido proporciona un mayor rendimiento por vatio [5].

Para poder mitigar esta limitación, se han propuesto los procesadores multicore *asimétricos* o AMPs [4, 1] (*Asymmetric Multicore Processors*). Un procesador de este tipo está constituido por un número considerable de cores simples, potencialmente lentos pero con un consumo reducido, y por unos pocos cores más complejos, más rápidos aunque con mayor consumo energético, todos ellos exponiendo un mismo repertorio de instrucciones. Ésta es una aproximación semejante al IBM Cell/BE [20], el ejemplo más destacado a nivel comercial de multicore heterogéneo, pero soportando explícitamente un repertorio de instrucciones común en todos los cores para simplificar el desarrollo de software, que es uno de los principales desafíos de sistemas como Cell [21].

Investigaciones previas en el área de diseño de AMPs concluyen que la integración de dos tipos distintos de core en el sistema permite una utilización muy eficiente del área del procesador, ofreciendo mejores prestaciones a las aplicaciones que los diseños simétricos [1, 3]. Además, el hecho de integrar sólo dos tipos de core ("rápidos" y "lentos") en la plataforma simplifica de forma considerable el diseño del software de sistema. No obstante, se espera que los fabricantes de hardware construyan sistemas AMP con distintos rangos de asimetría en rendimiento entre los cores, con el objetivo de satisfacer las demandas de distintos sectores del mercado [16].

El hecho de disponer de cores con distinto rendimiento y requisitos de área y consumo en un mismo procesador permite a los sistemas AMP ofrecer lo mejor de ambos mundos. Por un lado, los cores complejos del sistema contribuyen a mejorar el rendimiento de las aplicaciones puramente secuenciales, ya que estas aplicaciones no explotan el paralelismo a nivel de hilo que se derivaría de la ejecución en abundantes cores simples. Por otra parte, las aplicaciones paralelas que escalan a un gran número de cores pueden ejecutarse muy eficientemente asignando múltiples hilos de ejecución a los numerosos cores de bajo consumo.

Al contrario que los diseños multicore simétricos, los AMPs permiten la aceleración de las aplicaciones paralelas cuya escalabilidad está limitada por fases de ejecución secuencial [5]. Este potencial puede llevarse a cabo de manera transparente mediante la detección de las fases secuenciales o con paralelismo limitado, y la aceleración de éstas mediante cores complejos, con gran rendimiento serie [4].

Aunque el diseño de AMPs ha sido ampliamente investigado [1, 88, 23], y diversos estudios teóricos o basados en simulación muestran sus beneficios potenciales [5, 2, 13], la planificación de procesos/hilos en este tipo de arquitecturas no ha sido analizada aún con detalle.

En esta tesis nos hemos centrado en el diseño de algoritmos de planificación de procesos en el sistema operativo (SO) para AMPs. Las estrategias propuestas intentan maximizar el rendimiento global de estos sistemas mediante la explotación de diversas técnicas de especialización de cores (que presentaremos en la sección A.2.1), para así trasladar los beneficios de los sistemas asimétricos de manera transparente a las aplicaciones.

A.2. Explotando el potencial de los sistemas AMP

Los sistemas multicore asimétricos permiten *especializar* cada tipo de core para el tipo de aplicación o cómputo de la carga de trabajo que obtenga de él un mayor rendimiento por vatio. Trabajos de investigación previos han mostrado que la aplicación de distintas técnicas de especialización de cores contribuye a maximizar la eficiencia de estas arquitecturas para cargas de trabajo diversas [1, 3, 2, 13, 5]. Lamentablemente, el hardware asimétrico no explota estas técnicas por sí mismo, sino que es tarea del sistema operativo (SO) y/o del *runtime system* la extracción de los beneficios asociados a la especialización de cores.

El principal problema asociado a esta tarea es que los SOs de propósito general existentes no garantizan un uso eficiente de sistemas que integran cores de distinto tipo, y por tanto deben ser sometidos a un minucioso proceso de rediseño. Más concretamente, se deben incorporar en el software de sistema nuevas estrategias de planificación para llevar a cabo asignaciones de aplicaciones a los distintos tipos de cores, en función de las propiedades de éstas y teniendo en cuenta las características microarquitectónicas de los cores.

En esta sección pretendemos arrojar luz sobre estos problemas. Para ello, procedemos a continuación a la descripción de las técnicas de especialización de cores más relevantes que han sido propuestas en la literatura, y enumeramos los principales desafíos que tuvimos que afrontar al explotar estas técnicas mediante algoritmos de planificación implementados en el SO.

A.2.1. Técnicas de especialización de cores

Hasta la fecha se han propuesto distintas técnicas de especialización de cores que permiten identificar los tipos de aplicaciones que realizan un uso más eficiente (mayor rendimiento por vatio) de cores simples y complejos en un AMP [3, 2, 13, 4]. La mayor parte de estas técnicas son variaciones de dos formas "primitivas" de especialización: especialización de paralelismo a nivel de instrucción (ILP) y especialización de paralelismo a nivel de hilo (TLP).

Especialización de ILP

En entornos de ejecución multiprogramados, los hilos de ejecución presentan diferentes propiedades y distintas necesidades de uso de recursos hardware. En sistemas asimétricos, estas diferencias se traducen en que las aplicaciones exhiben distintos *factores de ganancia*: el beneficio relativo que cada aplicación obtiene de ejecutar en un core complejo con respecto a uno simple. La especialización de ILP explota esta diversidad para maximizar el rendimiento global del sistema AMP.

Para ilustrar la diversidad en los factores de ganancia de las aplicaciones realizamos un estudio en un sistema asimétrico emulado, donde los cores presentan la misma

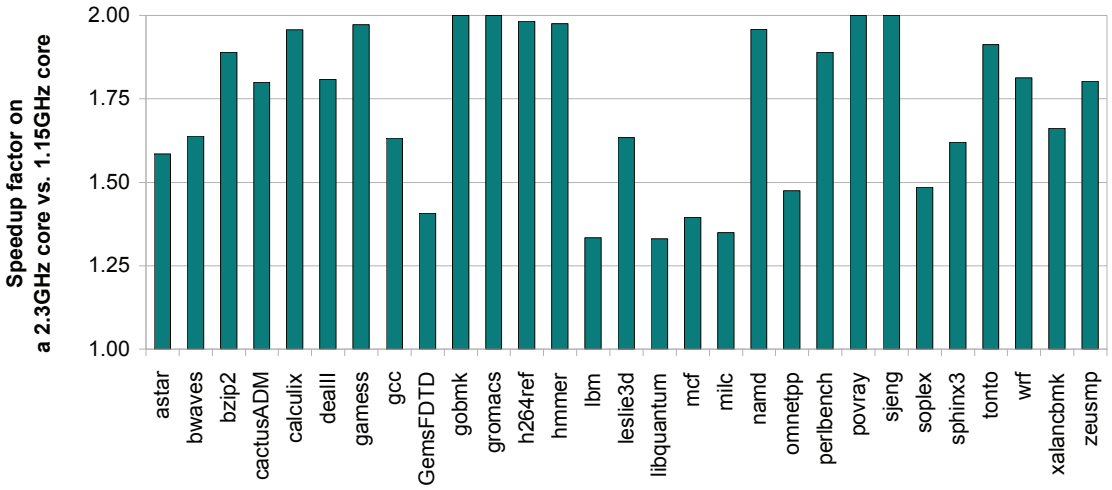


Figura A.1: Factor de ganancia experimentado por los *benchmarks* SPEC CPU2006 al ejecutar en un core "rápido" (2.3GHz) con respecto a un core "lento" (1.15GHz) en un sistema AMP emulado.

microarquitectura pero difieren en frecuencia de trabajo[1]. La figura A.1 muestra el factor de ganancia que los distintos benchmarks de la suite SPEC CPU2006 obtienen en este sistema asimétrico.

Algunas de estas aplicaciones experimentan un factor de ganancia de 2×, que es proporcional a la diferencia de frecuencias entre los cores " rápidos" y "lentos" del sistema. Estos programas hacen un uso muy eficiente del *pipeline* y de las unidades funcionales del procesador debido a su alto paralelismo a nivel de instrucción, que permite el lanzamiento a ejecución de múltiples instrucciones en paralelo sin dejar el procesador inactivo. Cabe también destacar que estas aplicaciones suelen mostrar una excepcional localidad en sus patrones de acceso a memoria, lo cual se traduce en una frecuente reutilización de los datos en la jerarquía cache y en infrecuentes accesos a memoria. Nos referiremos a estos programas como *intensivos en CPU*.

En el extremo opuesto se encuentran aquellas aplicaciones que experimentan sólo una pequeña fracción del máximo factor de ganancia alcanzable. Por ejemplo, los *benchmarks* `lbm` y `libquantum` aceleran solamente un 33 % al ejecutar en un core "rápido" con respecto a uno "lento". Estos son claros ejemplos de aplicaciones *intensivas en memoria* que provocan frecuentes paradas del *pipeline*, ya que pasan una parte significativa de su tiempo de ejecución transfiriendo datos desde memoria principal. Por este motivo, el incremento en la frecuencia de la CPU no se traduce en un rendimiento proporcional a éste para aplicaciones intensivas en memoria.

En escenarios multiaplicación, resulta más beneficioso para el rendimiento global del sistema (y para el consumo de energía) la ejecución de las aplicaciones intensivas en

[1]La emulación del sistema asimétrico en este ejemplo se llevó a cabo reduciendo las frecuencias de algunos de los cores de una plataforma multicore simétrica basada en procesadores AMD Opteron "Barcelona". Para ello se utilizaron las extensiones de escalado de voltaje y frecuencia (DVFS) de los cores incluidas en dicha plataforma. Los cores "rápidos" operan a la máxima frecuencia soportada (2.3 Ghz.), mientras que los cores lentos operan al mínimo nivel DVFS (1.15 Ghz.).

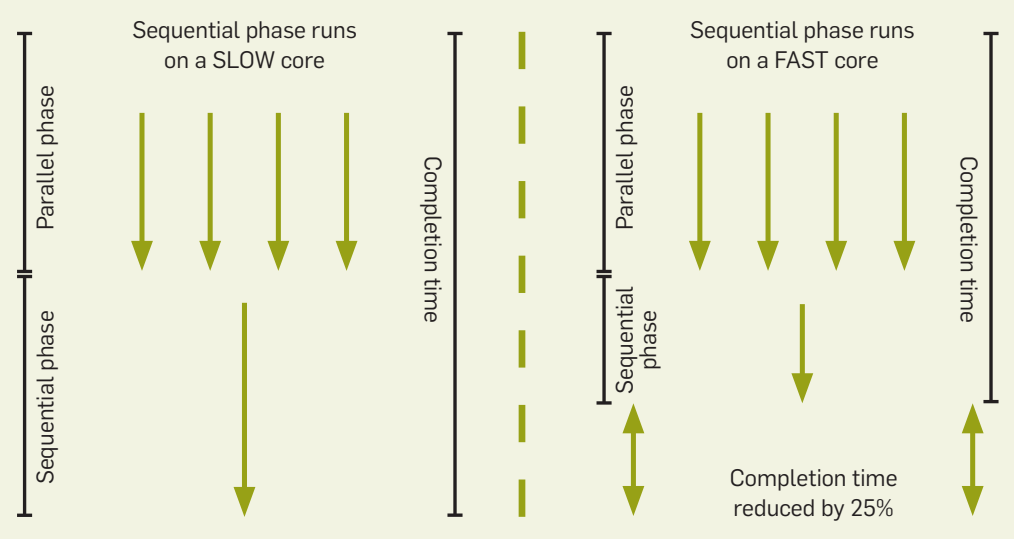


Figura A.2: Aceleración de fases secuenciales de aplicaciones paralelas en AMPs.

CPU en cores rápidos y complejos, y la asignación de aquellas intensivas en memoria a cores simples de consumo reducido. Esto garantiza la explotación efectiva de la diversidad de paralelismo a nivel de instrucción o especialización de ILP.

Trabajos de investigación recientes realizados conjuntamente por la Universidad de California en San Diego y HP [1, 3], muestran que el empleo de esta técnica de especialización de cores permite a los sistemas asimétricos ofrecer una mejora en el rendimiento de hasta un 63 % con respecto a sistemas multicore simétricos de área y consumo de energía similares.

Especialización de TLP

La especialización de TLP explota la diversidad de paralelismo a nivel de hilo (TLP) que exhiben las distintas aplicaciones. Esta diversidad hace referencia a las dos principales categorías en las que las aplicaciones pueden clasificarse en base a su TLP: aplicaciones *escalables* y *no escalables*. Las aplicaciones enmarcadas en la primera categoría utilizan mútiples hilos de ejecución de manera eficiente, de tal manera que incrementos en el número de hilos conllevan una notable reducción en el tiempo de ejecución o se traducen en un mayor trabajo realizado por unidad de tiempo. La segunda categoría abarca tanto las aplicaciones puramente secuenciales como aquellas que sólo escalan a un determinado número de cores. Encuadramos también en esta categoría a aquellas aplicaciones "híbridas" que alternan fases de ejecución multi-hilo escalables con fases secuenciales.

A diferencia de los sistemas multicore simétricos, los AMPs ofrecen un diseño optimizado para la ejecución eficiente de cargas de trabajo multiprogramadas incluyendo aplicaciones de ambas categorías. Las aplicaciones *escalables* pueden ejecutarse eficientemente en los numerosos cores simples de bajo consumo, explotando de este modo su alto grado de paralelismo a nivel de hilo. Por el contrario, las aplicaciones

secuenciales han de asignarse a cores complejos, pues derivan de éstos un mayor rendimiento. Este modo de uso de los distintos cores para la ejecución de los distintos tipos de aplicación se conoce como *especialización de TLP*.

El notable rendimiento secuencial ofrecido por los cores complejos de un AMP puede ser también explotado por las aplicaciones paralelas, ya que la ejecución de las fases secuenciales de éstas en cores de este tipo contribuye a la reducción de su fracción secuencial. Por ejemplo, consideremos una aplicación con una fracción secuencial que constituye un 50 % de su tiempo de ejecución. Supongamos que dicha aplicación se ejecuta en un sistema asimétrico donde los cores complejos son el doble de rápidos que los simples. En este escenario, ilustrado en la figura A.2, la ejecución de la totalidad de la región secuencial de la aplicación en un core complejo se traduce en una mejora de un 25 % en su rendimiento, según la ley de Amdahl [12]. En uno de sus trabajo teóricos, Hill y Marty demuestran que los sistemas asimétricos ofrecen un mayor rendimiento para las aplicaciones paralelas que los sistemas simétricos de área similar, siempre y cuando las fases secuenciales de éstas constituyan al menos un 5 % de su tiempo de ejecución [5].

A.2.2. Principales desafíos

Las técnicas de especialización previamente descritas permiten maximizar la explotación de los sistemas asimétricos, ofreciendo un mayor rendimiento por vatio. Para explotar el potencial de estas técnicas y trasladar su beneficio a las aplicaciones de manera transparente, es preciso incorporar el soporte adecuado en el sistema operativo y el *runtime system*.

Durante el diseño de los algoritmos de planificación de procesos/hilos propuestos en esta tesis, detectamos una serie de factores que dificultan tanto su implementación y efectividad, como el proceso de estudio de las ventajas e inconvenientes que presentan las distintas propuestas.

Aproximación de los factores de ganancia

El principal desafío que hemos encontrado durante el desarrollo de esta tesis es la aproximación precisa de los *factores de ganancia* de las aplicaciones; en otras palabras, determinar el beneficio relativo que éstas obtienen al ejecutarse en cores complejos con respecto a cores simples. Como comentamos previamente, los algoritmos que explotan la especialización de ILP asignan hilos con mayores factores de ganancia a cores complejos, ya que estos hilos son los que suelen hacer un uso más eficiente de este tipo de cores. Dos aproximaciones han sido propuestas por la comunidad científica para determinar estos factores.

La primera aproximación requiere ejecutar las aplicaciones *tanto en cores simples como complejos* para monitorizar el número de instrucciones retiradas por ciclo (IPC) en cada core. El factor de ganancia de cada aplicación se aproxima mediante el cociente de IPCs asociados a ésta en cores complejos y simples [2, 3].

La segunda aproximación está basada en la estimación del factor de ganancia a partir de medidas de rendimiento de la aplicación *en un solo tipo de core*, obtenidas mediante *profiling offline* o usando los contadores hardware del procesador durante la ejecución. Aunque las estimaciones son, a *priori*, menos precisas que los factores de ganancia obtenidos mediante medición directa, la estrategia basada en estimación evita problemas asociados a la necesidad de medir el rendimiento tanto en cores simples como complejos. En la sección A.5 mencionaremos algunos de estos problemas.

Detección de fases secuenciales en aplicaciones paralelas

Para explotar el potencial de la especialización de TLP, el planificador de procesos debe detectar de manera efectiva las fases paralelas y secuenciales de las aplicaciones en ejecución. La manera más directa para lograr este objetivo es monitorizar el número de hilos *activos* o listos para ejecutar en la aplicación. Si una aplicación posee un gran número de hilos activos, es muy probable que ésta esté ejecutando una fase con gran paralelismo. En cambio, aplicaciones con un solo hilo activo ejecutan código puramente secuencial.

Una buena propiedad de esta técnica de detección de fases con distinto paralelismo es que puede implementarse muy eficientemente en la mayoría de los sistemas operativos de propósito general existentes, ya que el contador de hilos activos en una aplicación es visible a nivel del *kernel* del SO[2].

Lamentablemente, el uso de esta heurística de detección de fases secuenciales no proporciona resultados satisfactorios en algunos casos. Esto se debe a que la aplicación podría estar ejecutando fases paralelas no escalables, pero usando aún un gran número de hilos activos [35, 36]. En esta tesis analizamos los escenarios en los que ésto ocurre, describimos las soluciones propuestas en la literatura y proponemos una solución a nivel de *runtime system* para la detección efectiva de fases secuenciales.

Disminución de la sobrecarga introducida por las migraciones de hilos

Un importante desafío a la hora de implementar algoritmos de planificación para AMPs es disminuir la sobrecarga asociada a las migraciones de hilos entre cores. Todo algoritmo de planificación que explote técnicas de especialización de cores requiere de las migraciones de hilos para lograr sus objetivos. Por ejemplo, si una aplicación transita de una fase de ejecución paralela a una secuencial, los algoritmos que explotan especialización de TLP deben garantizar que el hilo que ejecuta este código secuencial está asignado a un core complejo. De este modo, si este hilo se está ejecutando en un core simple, el planificador debe llevar a cabo una migración de éste a un core complejo para así garantizar la aceleración efectiva de la fase secuencial de la aplicación [4].

[2] En la actualidad la mayoría de sistemas operativos realizan una correspondencia uno a uno entre hilos a nivel de usuario e hilos a nivel de *kernel*.

Lamentablemente, las migraciones pueden resultar bastante costosas, sobre todo si el core origen y el destino están ubicados en distintos dominios de la jerarquía de memoria[3]. En arquitecturas NUMA, el coste asociado a accesos a bancos remotos de memoria contribuye a agravar esta situación, ya que el coste de las migraciones puede ser aún más elevado [17].

Mediante nuestra evaluación experimental, hemos detectado que la simple restricción del número de migraciones puede, en ocasiones, hacer decrecer el beneficio obtenido por los algoritmos de planificación para AMPs. Además de hacer al planificador consciente de la sensibilidad a las migraciones específica de cada aplicación, creemos que sería beneficioso el estudio de formas de reducir el coste asociado a las migraciones. En esta tesis mostramos diferentes topologías de sistemas asimétricos que permiten reducir este coste de manera significativa. También proporcionamos directivas de diseño del planificador del sistema operativo para mitigar de manera efectiva los efectos negativos de las migraciones mediante la explotación de la topología de la plataforma.

Evaluación de las propuestas de planificación

A pesar del creciente interés por parte de los principales fabricantes de hardware por los sistemas asimétricos con repertorio común de instrucciones –como dejan patentes las últimas contribuciones de Intel [8, 16, 24, 17] y HP [1, 3] a la literatura–, este tipo de procesadores aún no está siendo fabricado. Este hecho obliga a la creación de plataformas de pruebas, empleando técnicas de emulación o simulación de la asimetría, para realizar un estudio exhaustivo de los requisitos y desafíos del software de sistema para futuras plataformas asimétricas.

Como indicamos previamente, los algoritmos de planificación propuestos en esta tesis persiguen maximizar el rendimiento en sistemas AMP para muy diversas cargas de trabajo, potencialmente constituidas de aplicaciones paralelas. Para llevar a cabo una evaluación exhaustiva de estos algoritmos, nuestra plataforma experimental debía cumplir dos requisitos. En primer lugar, la plataforma debía estar basada en herramientas disponibles para investigación en el ámbito académico. En segundo lugar, esta plataforma debía permitir la emulación eficiente de un gran número de cores (para ejecutar aplicaciones paralelas) y facilitar el estudio de implementaciones de los algoritmos de planificación en un sistema operativo real.

De todas las técnicas de emulación y simulación disponibles para la comunidad académica, el empleo de técnicas de escalado de voltaje y frecuencia (DVFS) [27, 44, 45] para la creación de plataformas asimétricas resultó satisfacer plenamente nuestros requisitos. Esta técnica se basa en la introdución de la asimetría en rendimiento mediante la disminución de la frecuencia de trabajo de algunos de los cores en plataformas multicore simétricas.

La principal ventaja ofrecida por el uso de DVFS es la emulación eficiente de sistemas multicore asimétricos con repertorio común de instrucciones. Esto permite

[3]En este contexto, un dominio de la jerarquía de memoria se define como un grupo de cores compartiendo una cache de último nivel.

llevar a cabo una evaluación exhaustiva con un número elevado de cores para la ejecución simultánea de múltiples hilos. La rápida emulación nos ha permitido estudiar en detalle diferentes técnicas de detección de fases de ejecución secuencial en diversas cargas de trabajo y aceleración automática de estas fases en AMPs.

Aunque esta técnica de emulación sólo permite evaluar sistemas con cores que poseen la misma microarquitectura, el tipo de asimetría que se consigue emular permite a las aplicaciones exhibir una gran diversidad en factores de ganancia (como muestra la figura A.1). Esta diversidad hace posible el estudio de algoritmos que explotan la especialización de ILP.

A.3. Algoritmos de planificación propuestos

Los algoritmos de planificación de procesos propuestos en esta tesis han sido diseñados explícitamente para sistemas asimétricos donde los cores difieren en rendimiento pero ofrecen un repertorio de instrucciones común. Todas nuestras estrategias de planificación persiguen dos objetivos comunes. En primer lugar, éstas intentan maximizar el rendimiento global del sistema mediante la explotación de las técnicas de especialización de cores descritas en la sección A.2.1. En segundo lugar, estas estrategias trasladan los beneficios de los sistemas asimétricos directamente a las aplicaciones sin requerir su modificación o recompilación

Los principales algoritmos de planificación propuestos en esta tesis son los planificadores HASS, PA y CAMP. Antes de proseguir con una breve descripción de los mismos, debemos aclarar que la implementación de estos algoritmos no está aún preparada para hacer frente a los problemas asociados a la existencia de asimetría funcional entre los cores (diferencias en repertorios de instrucciones). Investigaciones recientes de los principales fabricantes de hardware indican que la asimetría funcional podría estar presente a pequeña escala en los futuros sistemas asimétricos, de tal manera que los cores del sistema sean idénticos en prácticamente todos los aspectos del repertorio de instrucciones pero difieran en un pequeño subconjunto de éste [8]. No obstante, los algoritmos que proponemos pueden extenderse de manera sencilla con técnicas de *fault-and-migrate*, que evitan errores en tiempo de ejecución originados por diferencias funcionales entre los cores [16].

A.3.1. HASS

El planificador HASS (*Heterogeneity-Aware Signature-Supported scheduler*) maximiza el rendimiento global del sistema asimétrico gracias a la explotación de la diversidad en las propiedades microarquitectónicas que exhiben las distintas aplicaciones en una carga de trabajo. HASS está basado en las *firmas arquitectónicas*: resúmenes compactos de las características de las aplicaciones (patrones de acceso a memoria, paralelismo a nivel de intrucción, ...). La información incluida en la firma arquitectónica de cada aplicación permite a HASS determinar su *factor de ganancia*.

La primera implementación de HASS (versión estática o HASS-S) –basada en firmas arquitectónicas obtenidas mediante *profiling offline*– fue propuesta en uno de nuestros trabajos previos liderado por D. Shelepov [22]. En esta tesis proponemos una versión dinámica y más versátil de HASS (HASS-D) que se basa en la contrucción de firmas arquitectónicas en tiempo de ejecución. Mediante una evaluación exhaustiva de HASS-S y HASS-D, demostramos que ambos algoritmos superan las principales limitaciones de algoritmos previamente propuestos, que se basan en diferentes técnicas para determinar los factores de ganancia de las aplicaciones.

A.3.2. PA

El algoritmo de planificación PA (*Parallelism Aware*) es el primer algoritmo a nivel de sistema operativo específicamente diseñado para acelerar automáticamente las fases secuenciales de las aplicaciones paralelas en AMPs. Para conseguir sus objetivos, PA detecta fases de ejecución puramente secuenciales o con paralelismo limitado en una aplicación y asigna los hilos que ejecutan estas fases a cores complejos. Por el contrario, hilos de ejecución ejecutando fases de la aplicación que escalan a un gran número de cores se ejecutan en los abundantes cores simples y de consumo reducido.

El planificador PA está equipado con un mecanismo eficiente de detección de fases secuenciales y paralelas en las aplicaciones. Como indicamos en la sección A.2.2, estas fases pueden detectarse de manera sencilla por el SO cuando los hilos bloqueados en primitivas de sincronización utilizan espera bloqueante. En caso de que los hilos no se bloqueen en estas circunstancias, realizando en su lugar esperas activas en modo usuario (*spin*), esta técnica no garantiza buenos resultados, ya que estos hilos permanecen activos sin realizar trabajo útil.

Para hacer frente a estos escenarios, PA ofrece un conjunto de extensiones de *runtime* (PA-RTX) que permiten a la librería de hilos notificar al planificador del SO qué hilos están realizando esperas activas, y cuáles son más propensos a la ejecución de fases explícitamente secuenciales en la aplicación. En esta tesis hemos extendido el *runtime system* de una conocida implementación de OpenMP con las extensiones de PA. Mediante un completo estudio experimental de dicha implementación, mostramos que PA consigue detectar de manera efectiva las fases con limitado paralelismo a nivel de hilo de las aplicaciones, sin requerir su modificación.

A.3.3. CAMP

Algoritmos previamente propuestos en la literatura, así como nuestros algoritmos HASS y PA, resultan efectivos sólamente para ciertas cargas de trabajo. Por ejemplo, estrategias que realizan asignaciones de hilos únicamente en base a sus factores de ganancia, como HASS, garantizan una utilización eficiente de los sistemas AMP cuando planifican cargas de trabajo constituidas únicamente por aplicaciones secuenciales. En esta tesis hemos demostrado experimentalmente que estos algoritmos ofrecen un mayor rendimiento que PA para este tipo de cargas de trabajo,

pero la situación se revierte en escenarios con aplicaciones paralelas. La razón de este comportamiento es que estos algoritmos sólo tienen en cuenta los factores de ganancia de las aplicaciones o bien su paralelismo a nivel de hilo (TLP), pero nunca ambos.

Para hacer frente a estas limitaciones, procedimos a diseñar el planificador CAMP (*a Comprehensive scheduler for Asymmetric Multicore Processors*), la propuesta final de esta tesis. CAMP determina a qué tipo de core ha de asignarse cada hilo de ejecución teniendo en cuenta su *factor de utilidad*: métrica que aproxima la ganancia relativa que la aplicación a la que este hilo pertenece obtendría de utilizar de manera exclusiva todos los cores complejos del sistema asimétrico, con respecto a una ejecución donde sólo podrían utilizarse cores simples. Para aproximar el factor de utilidad de una aplicación, CAMP utiliza información del paralelismo a nivel de instrucción (factor de ganancia) y a nivel de hilo (TLP) de la misma.

CAMP garantiza que hilos con un mayor factor de utilidad se asignan a cores complejos y el resto a cores simples. Esta "regla" determina indirectamente que las regiones de las aplicaciones con limitado paralelismo a nivel de hilo y un elevado factor de ganancia gocen de una *mayor prioridad* para ejecutar en cores complejos. Esto contribuye a mejorar el rendimiento global del sistema asimétrico de manera significativa, ya que esas regiones de las aplicaciones son las que explotan este tipo de cores más eficientemente.

A.4. Principales aportaciones

Las principales aportaciones de esta tesis son las siguientes:

- Los algoritmos de planificación propuestos han sido implementados en un sistema operativo real y evaluados exhaustivamente en hardware multicore real convertido en asimétrico mediante técnicas de escalado de voltaje y frecuencia (DVFS). Ninguno de estos algoritmos requiere cambios en las aplicaciones para alcanzar sus objetivos, sino modificaciones en el sistema operativo. El hecho de habernos centrado en el estudio de implementaciones reales de los algoritmos nos ha llevado a interesantes descubrimientos que habrían sido imposibles de detectar mediante simulación. Por este motivo, esta tesis arroja luz sobre los principales desafíos que los desarrolladores de sistemas operativos tendrán que afrontar en el futuro para garantizar una máxima explotación de los sistemas AMPs.

- Los algoritmos de planificación para AMPs propuestos en trabajos previos y evaluados mediante simulación determinaban los *factores de ganancia* de los hilos de ejecución (beneficio relativo aportado por un core rápido y complejo con respecto a uno simple) realizando medidas de rendimiento en *ambos* tipos de core [3, 2]. Durante la evaluación de implementaciones reales de estos algoritmos –realizadas en esta tesis–, detectamos que esta metodología para

determinar los factores de ganancia presenta serias limitaciones para ser aplicada en la práctica. La detección de problemas de este tipo junto con el diseño de algoritmos de planificación que no son susceptibles a estas deficiencias son aportaciones clave de esta tesis.

- La beneficios que los AMPs ofrecen a las aplicaciones paralelas para mitigar sus cuellos de botella secuenciales han sido ampliamente demostrados mediante estudios analíticos o prototipos de planificación a nivel de usuario [4, 5, 23]. En esta tesis hemos diseñado e implementado el primer planificador a nivel de sistema operativo que explota este potencial de los AMPs y traslada sus beneficios automáticamente a las aplicaciones. Nuestro estudio nos permitió identificar las principales barreras existentes a nivel del SO asociadas a la extracción de este potencial, y determinar hasta qué punto el sistema operativo puede detectar por sí mismo las fases con limitado paralelismo a nivel de hilo en las aplicaciones. En nuestro estudio también demostramos que la interacción entre el *runtime system* y el SO es primordial para explotar esta atractiva característica de los sistemas multicore asimétricos.

- Estudios teóricos previos concluyen que tanto el paralelismo a nivel de instrucción (ILP) de una aplicación como su paralelismo a nivel de hilo (TLP) determinan el beneficio relativo que ésta obtiene al ejecutar en cores complejos con respecto cores simples [1, 5]. En esta tesis proponemos *el factor de utilidad*, una métrica que permite aproximar este beneficio relativo en función del TLP e ILP de la aplicación. El factor de utilidad es el elemento principal de nuestro planificador CAMP.

A.5. Conclusiones

Trabajos de investigación recientes han mostrado los beneficios potenciales que los procesadores multicore asimétricos con repertorio común de instrucciones (AMPs) ofrecen con respecto a sus equivalentes simétricos [1, 5]. Por ello, es probable que próximas generaciones de procesadores multicore estén constituidos por cores que difieran en rendimiento y consumo de energía, pero que soporten un repertorio de instrucciones común que permita simplificar el desarrollo de software.

Numerosos estudios en este área concluyen que este potencial de los sistemas AMP puede extraerse mediante la aplicación de técnicas de *especialización de cores*. Estas técnicas permiten identificar el tipo de aplicación/cómputo presente en la carga de trabajo que garantiza un uso más eficiente de cada tipo de core en términos de rendimiento y consumo [2, 4, 13]. La principal barrera en este aspecto es que la especialización de los cores no es explotada de manera transparente por el hardware, sino que es tarea del software de sistema (sistema operativo y/o *runtime system*) la aplicación de estas técnicas para así trasladar los beneficios potenciales de los AMPs a las aplicaciones sin requerir su modificación ni recompilación.

Aunque el diseño de los sistemas multicore asimétricos ha sido ampliamente investigado [1, 88, 23] y sus beneficios potenciales se han hecho patentes mediante estudios

teóricos o basados en simulación [5, 2, 13], no se ha llevado a cabo hasta la fecha un estudio exhaustivo del soporte necesario en un sistema operativo real que permita hacer realidad estos beneficios de manera transparente a las aplicaciones.

El principal objetivo de esta tesis ha sido investigar cómo y hasta qué punto, las estrategias de especialización de cores pueden aplicarse mediante planificación de procesos en el sistema operativo. Para ello, hemos propuesto tres algoritmos de planificación para AMPs (HASS, PA and CAMP), cuya implementación en un sistema operativo real ha sido evaluada de manera exhausiva en plataformas multicore asimétricas emuladas mediante la aplicación de técnicas de escalado de voltaje y frecuencia (DVFS).

Nuestro algoritmo de planificación HASS (*Heterogeneity-Aware Signature-Supported scheduler*) garantiza un uso eficiente del sistema asimétrico gracias a la explotación de la diversidad en las propiedades microarquitectónicas que exhiben las distintas aplicaciones en una carga de trabajo. Para ello, HASS identifica aquellas aplicaciones que experimentan un mayor *factor de ganancia*. Las aplicaciones con mayores factores de ganancia se asignan a cores complejos y el resto a cores simples. El factor de ganancia de cada aplicación se obtiene mediante estimación y en función de sus propiedades microarquitectónicas. En esta tesis hemos analizado dos versiones de HASS, HASS-S y HASS-D, que monitorizan distintas métricas de rendimiento de las aplicaciones usando *profiling offline* y contadores hardware del procesador durante la ejecución, respectivamente. Las principales conclusiones que obtuvimos de nuestro estudio son las siguientes:

- Los beneficios de los algoritmos de planificación para sistemas asimétricos son especialmente significativos para cargas de trabajo donde existe gran disparidad entre los factores de ganancia de las aplicaciones, y en sistemas donde los cores complejos ofrecen mucho mayor rendimiento que los cores simples. En concreto, nuestros resultados experimentales revelan que, para todas las cargas de trabajo investigadas, HASS consigue un mejor rendimiento que el planificador por defecto del sistema, que no es consciente de la presencia de distintos cores en la plataforma.

- Superando nuestras expectativas, ambas versiones de HASS (estática y dinámica) mejoran con creces el rendimiento del algoritmo IPC-Driven, previamente propuesto por Becchi y Crowley en [2]. Durante nuestra evaluación experimental, detectamos que este algoritmo esta sujeto a imprecisiones en el cálculo de los factores de ganancia así como a sobrecargas significativas. El origen de estos problemas de IPC-Driven reside en su necesidad de medir directamente el rendimiento en cores de distinto tipo para determinar los factores de ganancia, en lugar de recurrir para ello a técnicas de estimación como HASS.

- Por último, concluimos que la versión estática de HASS (HASS-S) –basada en la estimación de los factores de ganancia usando información recabada mediante *profiling offline* – introduce menor sobrecarga que la versión dinámica (HASS-D), que, por el contrario, monitoriza rendimiento de las aplicaciones en

tiempo de ejecución. Por esta razón, HASS-S ofrece un rendimiento ligéramente superior que HASS-D para la mayoría de las cargas de trabajo investigadas. No obstante, en escenarios en los que la información recabada estáticamente no esté disponible (p. ej., no está incluida en el ejecutable de la aplicación) o cuando ésta no sea suficientemente representativa durante la ejecución (p. ej., la aplicación posee múltiples y muy diversas fases de ejecución), la monitorización *online* del rendimiento permite solucionar este problema. Por lo tanto, la versión dinámica de HASS constituye una estrategia de planificación más robusta para sistemas asimétricos de propósito general.

En esta tesis también hemos analizado en detalle las ventajas que los procesadores multicore asimétricos ofrecen a las aplicaciones paralelas con cuellos de botella secuenciales. Como comentamos previamente, los cores complejos del sistema pueden utilizarse de manera oportunista para acelerar las fases con paralelismo limitado de las aplicaciones, ya que estos cores ofrecen un mayor rendimiento secuencial que los abundantes cores simples de bajo consumo. Para llevar a cabo este análisis utilizamos nuestro algoritmo de planificación PA (*Parallelism Aware*), la primera propuesta de planificación a nivel de sistema operativo que explota esta destacada característica de los sistemas asimétricos. Los resultados obtenidos nos han llevado a concluir lo siguiente:

- Algoritmos simples de planificación que maximizan la utilización de los cores complejos del sistema, como BusyFCs [14, 17] o RR [2], consiguen acelerar de manera efectiva las fases secuenciales de las aplicaciones paralelas cuando sólo una aplicación se ejecuta en el sistema. No obstante, en escenarios multiaplicación y en aquellos casos en los que hilos de ejecución realizan espera activa (*spin*) durante fases de sincronización en lugar de espera bloqueante, los algoritmos de planificación mencionados previamente no logran su objetivo y PA es capaz de superarlos, obteniendo una mejora en el rendimiento de hasta un 40 %.

- PA consigue detectar la mayor parte de las fases secuenciales de las aplicaciones mediante la monitorización del número de hilos activos en el proceso asociado a la misma. En algunos casos, sin embargo, los hilos bloqueados por motivos de sincronización realizan esperas activas durante cortos periodos de tiempo y, por tanto, el sistema operativo no puede detectar las fases secuenciales teniendo en cuenta simplemente el número de hilos activos. Para hacer frente a esta limitación, diseñamos un conjunto de extensiones de *runtime system* (PA-RTX) que ofrecen un interfaz simple de operaciones permitiendo a la librería de gestión de hilos notificar al planificador del sistema operativo qué hilos están realizando esperas activas y no trabajo útil. Los resultados obtenidos indican que PA consigue beneficios adicionales gracias a estas extensiones, lo cual muestra el papel esencial de la interacción entre el *runtime system* y el sistema operativo para la detección efectiva de fases con paralelismo limitado en software paralelo.

- Las migraciones de hilos entre cores de distinto tipo son una herramienta clave para los algoritmos de planificación en sistemas asimétricos. Sin embargo, estas

migraciones pueden resultar especialmente costosas en caso de que el core origen no se encuentre en el mismo *dominio* de la jerarquía de memoria que el core destino. El estudio de mecanismos para hacer las migraciones menos costosas, nos llevó a concluir que la sobrecarga asociada a éstas puede reducirse de manera significativa en sistemas asimétricos diseñados de tal modo que hubiese un core rápido en cada dominio de la jerarquía de memoria[4] (por ejemplo, por cada grupo de cores lentos compartiendo la cache de último nivel). La reducción en el coste medio de las migraciones es factible siempre y cuando el planificador del sistema evite migraciones de hilos entre distintos dominios de memoria cuando sea posible. Creemos que esta conclusión será de gran utilidad para los diseñadores de futuros sistemas asimétricos.

La evaluación experimental de los algoritmos HASS y PA llevada a cabo en esta tesis muestra que ambos maximizan el rendimiento de los sistemas asimétricos para diversas cargas de trabajo. Estas estrategias de planificación son claros ejemplos de las dos categorías de algoritmos de planificación para AMPs propuestas hasta el momento en la literatura. Los primeros explotan la diversidad en el comportamiento a nivel microarquitectónico y de paralelismo a nivel de instrucción (ILP) que exhiben las aplicaciones [2, 3]. Los segundos, en cambio, tienen en cuenta el grado de paralelismo a nivel de hilo (TLP) a la hora de efectuar las asignaciones de aplicaciones a cores simples o complejos [4]. Lamentablemente, los planificadores encuadrados en cualquiera de estas categorías sólo explotan la información referente al ILP o al TLP de las aplicaciones pero nunca ambas. Por este motivo, sólo obtienen beneficios significativos para cargas de trabajo específicas.

Esta importante limitación nos motivó a proponer y diseñar CAMP (*Comprehensive scheduler for AMPs*). CAMP toma decisiones de planificación en base al *factor de utililidad* de las aplicaciones, métrica propuesta en esta tesis que aproxima, en función del ILP y TLP de una aplicación, el factor de ganancia global que ésta obtendría si se permite que sus hilos de ejecución utilicen todos los cores rápidos y complejos en el sistema[5], con respecto a una ejecución donde sólo se utilizan cores simples. Las principales conclusiones extraídas del análisis de CAMP son las siguientes:

- Los algoritmos que explotan la diversidad en paralelismo a nivel de instrucción (ILP) que exhiben las aplicaciones, como HASS, son incapaces de ofrecer un rendimiento similar a CAMP para cargas de trabajo que incluyen aplicaciones paralelas. Del mismo modo, algoritmos como PA –que explotan la información acerca del TLP de las aplicaciones– no consiguen superar a CAMP en escenarios constituidos únicamente por aplicaciones secuenciales y en aquellos escenarios donde las aplicaciones en ejecución exhiben un amplio rango de factores de ganancia.

[4]Las migraciones a *cores* dentro del mismo dominio de jerarquía de memoria no afectan normalmente al rendimiento de forma significativa [69].

[5]Para el computo de ésta ganancia asumimos que en caso de que el número de hilos supere el número de cores complejos, sólo tantos hilos como cores complejos se asignan a éstos mientras que los hilos restantes se asignan a cores simples.

- Un elemento esencial que contribuye al éxito de CAMP es la eficiente técnica empleada para estimar los factores de ganancia de los distintos hilos de ejecución.

A pesar de las notables mejoras en el rendimiento global obtenidas por nuestros algoritmos en las plataformas AMP emuladas, creemos que estas mejoras serían aún más significativas en sistemas asimétricos reales. Es muy probable que los futuros sistemas AMP exhiban una diferencia de rendimiento más drástica entre cores simples y complejos, debida principalmente a diferencias en el *pipeline* o en la jerarquía cache. No obstante, como nuestros resultados apuntan, mayores diferencias en rendimiento se traducen en mejoras cada vez más significativas obtenidas por las estrategias de planificación para AMPs.

Ya que en los futuros sistemas AMP el acceso a memoria seguirá siendo uno de los principales factores que limitan el rendimiento [24], nuestros modelos de estimación de factores de ganancia –basados en el número de accesos al último nivel de cache y a memoria– servirán, al menos, como aproximación de primer orden para determinar estos factores.

Finalmente, concluimos que los sistemas multicore asimétricos constituyen una alternativa viable para ser la próxima generación de sistemas de computación de propósito general, siempre y cuando estén equipados con un soporte de sistema operativo y *runtime system* adecuado.

A.6. Trabajo futuro

En esta tesis nos hemos centrado en maximizar el rendimiento global en sistemas AMP mediante diversas estrategias de planificación. No obstante, numerosos factores y objetivos no explorados en nuestro trabajo contribuyen de manera adicional a un uso más eficiente de los sistemas AMP. Entre estos aspectos destacamos los siguientes:

- **Calidad de servicio y planificación basada en prioridades:** Las políticas de planificación exploradas en esta tesis se comportarían de manera *injusta* en escenarios donde las aplicaciones tengan distintas prioridades, o en casos donde sea necesario garantizar diversos niveles de calidad de servicio. No obstante, las estrategias de planificación que persigan estos objetivos pueden hacer un uso más eficiente del sistema asimétrico si tienen en cuenta los factores relativos de ganancia de las aplicaciones, su paralelismo a nivel de instrucción (ILP) y su grado de paralelismo a nivel de hilo (TLP). Por ejemplo, entre aplicaciones de alta prioridad intensivas en CPU, el planificador debería asignar preferentemente a cores rápidos las fases secuenciales de éstas, ya que destinar los "escasos" cores rápidos de un AMP para la ejecución de fases con alto grado de paralelismo a nivel de hilo no proporciona una mejora significativa en el rendimiento e incluso demanda un mayor consumo de energía [4].

- **Gestión de consumo de energía:** En nuestra evaluación experimental, hemos asumido que el factor de ganancia resultante de ejecutar una aplicación en un core complejo en lugar de en uno simple permanece constante durante fases estables[6] de la ejecución de un programa. Sin embargo, el sistema operativo o el propio hardware podrían desencadenar transiciones del estado de los cores a niveles de menor consumo –por ejemplo, debido a que los cores han estado inactivos durante un cierto tiempo–, que pueden traducirse en variaciones de estos factores de ganancia incluso en la situación descrita anteriormente. Para garantizar la utilización efectiva del sistema asimétrico en estas circunstancias, la planificación de procesos y la gestión de consumo deben realizarse de forma totalmente coordinada. Explorar el efecto conjunto de la asimetría en rendimiento de la plataforma y de las variaciones dinámicas del rendimiento de los cores durante la ejecución será una interesante dirección a tener en cuenta para trabajo futuro.

- **Planificación basada en el comportamiento de acceso a cache de las aplicaciones:** Otra posible vía para continuar nuestra investigación en este campo es estudiar la interacción entre las migraciones de hilos y el comportamiento de acceso a la jerarquía cache de las distintas aplicaciones. Numerosos estudios han demostrado que algunas aplicaciones son mucho más sensibles que otras a la migración de hilos entre cores debido a su naturaleza en los patrones de acceso a memoria [85, 86, 17], y que la planificación resulta mucho más compleja en escenarios donde algunos hilos de ejecución comparten datos mediante niveles de cache compartidos [87]. La explotación de esta información en algoritmos de planificación para AMPs, como CAMP, supondría un importante paso hacia el diseño de estrategias de planificación más robustas para un amplio espectro de sistemas asimétricos con topologías muy diversas.

- **Estimación de factores de ganancia en sistemas con gran asimetría en rendimiento:** La mayor parte de los modelos de estimación de factores de ganancia diseñados hasta la fecha han sido evaluados en sistemas donde los cores tienen idéntica microarquitectura pero difieren en frecuencia (como los investigados en esta tesis) o en tasa de retirada de instrucciones [24]. Sin embargo, no existe por el momento ningún modelo específicamente diseñado para sistemas asimétricos con profundas diferencias microarquitectónicas entre los distintos cores. Creemos que el primer paso para la consecución de este objetivo es la búsqueda de metodologías más versátiles que permitan, independientemente de las características que determinan las diferencias en rendimiento de los distintos cores, identificar las métricas específicas de cada plataforma que guíen al planificador en la aproximación de los factores de ganancia de manera más precisa.

[6]En este contexto, asumimos que una aplicación está en una *fase estable* cuando el número de hilos activos de ésta permanece estable y el número de instrucciones retiradas por ciclo (IPC) de cada uno de estos hilos permanece en un cierto rango determinado por un porcentaje de variación dado.

Appendix B

Publications

Conferences

- Juan Carlos Sáez, Alexandra Fedorova, Manuel Prieto and Hugo Vegas. Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors. In *Proc. of the 7th ACM International Conference on Computing Frontiers, (CF'10)*, pp. 31–40. Bertinoro, Italy. May 2010.

- Juan Carlos Sáez, Manuel Prieto, Alexandra Fedorova and Sergey Blagodurov. A Comprehensive Scheduler for Asymmetric Multicore Systems. In *Proc. of 5th ACM European Conference on Computer Systems (Eurosys '10)*, pp. 139–152. Paris, France. April 2010.

- Juan Carlos Sáez, Manuel Prieto, Alexandra Fedorova. OS Scheduling for ASISA Multicore Processors. In *Advanced Computer Architecture and Compilation for Embedded Systems (ACACES '09)*, pp. 179–182. Barcelona, Spain. July 2009.

- Juan Carlos Sáez, Jose Ignacio Gomez and Manuel Prieto. Improving Priority Enforcement via Non-Work-Conserving Scheduling. In *Proc. of the 37th International Conference on Parallel Processing (ICPP'08)*, pp. 99–106. Portland (Oregon), USA. 2008.

Journals

- Juan Carlos Sáez, Daniel Shelepov, Alexandra Fedorova and Manuel Prieto. Leveraging Workload Diversity Through OS Scheduling to Maximize Performance on Single-ISA Heterogeneous Multicore Systems", In *Journal of Parallel and Distributed Computing (JPDC)*, Vol. 71 , pp. 114–131. January 2011.

- Alexandra Fedorova, Juan Carlos Sáez, Daniel Shelepov and Manuel Prieto. Maximizing Power Efficiency with Asymmetric Multicore Systems. *Communications of the ACM*, Vol. 52 (12), pp 48–57. December 2009.
- Alexandra Fedorova, Juan Carlos Sáez, Daniel Shelepov and Manuel Prieto. Maximizing Power Efficiency with Asymmetric Multicore Systems. *ACM Queue*, Vol. 7(10), pp. 30–45. November 2009.
- Daniel Shelepov, Juan Carlos Sáez, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov and Viren Kumar. HASS: a Scheduler for Heterogeneous Multicore Systems. *ACM SIGOPS Operating Systems Review*, Vol. 43 (2), pp. 66–75. April 2009.

List of Figures

List of Tables

Bibliography

[1] Rakesh Kumar, Keith I. Farkas, and Norman Jouppi et al. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *Proc. of MICRO 36*, 2003.

[2] Michela Becchi and Patrick Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proc. of Computing Frontiers '06*, pages 29–40, 2006.

[3] Rakesh Kumar, Dean M. Tullsen, and Parthasarathy Ranganathan et al. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proc. of ISCA '04*, 2004.

[4] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law through EPI Throttling. In *Proc. of ISCA'05*, pages 298–309, 2005.

[5] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[6] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] Johan de Galas. The Quest for More Processing Power: is the Single Core CPU Doomed?, February 2005. `http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377`.

[8] Matt Gillespie. Preparing for The Second Stage of Multi-Core Hardware: Asymmetric (Heterogeneous) Cores. *Intel White Paper*, 2008.

[9] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, C. Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37(7):44–55, July 2004.

[10] David Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer*, 38:11–13, May 2005.

[11] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25:21–29, March 2005.

[12] G.M Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. of AFIPS Conference*, pages 483–48, 1967.

[13] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan, and Vanish Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 28(3):26–41, 2008.

[14] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH CAN*, 33(2):506–517, 2005.

[15] Soraya Ghiasi, Tom Keller, and Freeman Rawson. Scheduling for heterogeneous processors in server systems. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 199–210, New York, NY, USA, 2005. ACM.

[16] Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating System Support for Overlapping-ISA Heterogeneous Multi-Core Architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, 9-14 2010.

[17] Tong Li, Dan Baumberger, and David A. Koufaty et al. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. of SC '07*, pages 1–11.

[18] Muthu Venkatachalam, Prashant Chandra, and Raj Yavatkar. A highly flexible, distributed multiprocessor architecture for network processing. *Comput. Netw.*, 41(5):563–586, 2003.

[19] Intel Corporation. Intel® Atom™ Processor E6x5C Series-based Platform for Embedded Computing. *Platform Brief*, 2010.

[20] Michael Gschwind. The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262, 2007.

[21] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it Easier to Program The Cell Broadband Engine Processor. *IBM J. Res. Dev.*, 51(5):593–604, 2007.

[22] Daniel Shelepov, Juan Carlos Saez, and Stacey Jeffery et al. HASS: a Scheduler for Heterogeneous Multicore Systems. *ACM SIGOPS Operating System Review*, 43(2), 2009.

[23] Tomer Morad, Uri Weiser, and Avnoam Kolody. ACCMP – Asymmetric Cluster Chip Multi-Processing. *TR CCIT*, 2004.

[24] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias Scheduling in Heterogeneous Multi-core Architectures. In *Proc. of Eurosys '10*, 2010.

[25] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM TOCS*, 23(3):253–300, 2005.

[26] Ruud van der Pas. The OMPlab on Sun Systems. In *Proc. of IWOMP'05*, 2005.

[27] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISLPED '07: Proceedings of the 2007 international symposium on Low power electronics and design*, pages 38–43, New York, NY, USA, 2007. ACM.

[28] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. USENIX Summer 1990 Conference*, pages pp. 247–456, 1990.

[29] David Nellans, Rajeev Balasubramonian, and Erik Brunv. A case for increased operating system support in chip multiprocessors. In *In Proc. of 2nd IBM Watson P=ac 2*, 2005.

[30] Joshua A. Redstone, Susan J. Eggers, and Henry M. Levy. An analysis of operating system behavior on a simultaneous multithreaded architecture. *SIGOPS Oper. Syst. Rev.*, 34(5):245–256, 2000.

[31] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. *SIGARCH Comput. Archit. News*, 34(2):66–77, 2006.

[32] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[33] R.E. Grant and A. Afsahi. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., 25-29 2006.

[34] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[35] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. *SIGARCH CAM*, 36(1):277–286, 2008.

[36] Ryan Johnson, Manos Athanassoulis, Radu Stoica, and Anastasia Ailamaki. A New Look at the Roles of Spinning and Blocking. In *Proc. of DaMoN '09*, pages 21–26, 2009.

[37] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 2A and 3B: Instruction Set Reference. *http://www.intel.com/products/processor/manuals*, 2010.

[38] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 3A and 3B: System Programming Guide. *http://www.intel.com/products/processor/manuals*, 2010.

[39] Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors. *White paper*, November 2008.

[40] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26:52–60, 2006.

[41] Wind River. Simics: Transform your product development life cycle, 2010. `http://windriver.com/products/simics`.

[42] M.T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. pages 23 –34, apr. 2007.

[43] Dwayne Lee. OpenSPARC - A Scalable Chip Multi-Threading Design. In *VLSID '08: Proceedings of the 21st International Conference on VLSI Design*, page 16, Washington, DC, USA, 2008. IEEE Computer Society.

[44] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and I. Kolcu. Exploiting processor workload heterogeneity for reducing energy consumption in chip multiprocessors. In *In Design, Automation and Test in Europe*, pages 1158–1163, 2004.

[45] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[46] Intel Corporation. First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). *White paper*, April 2008.

[47] Wonyoung Kim, Meeta S. Gupta, Gu yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *in International Symposium on High-Performance Computer Architecture*, 2008.

[48] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core opteron processor. pages 102 –103, feb. 2007.

[49] Power-aware dispatcher. Sun's wikis, 2009. `http://wikis.sun.com/display/BluePrints/Power+Aware+Dispatcher`.

[50] Jim Held, Bautista Jerry, and Sean Koehl. From a Few Cores to Many: A Tera-scale Computing Research Overview. *Intel White Paper*, 2006.

[51] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[52] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-Independent Workload Characterization. *IEEE Micro*, 27(3):63–72, 2007.

[53] Timothy Sherwood, Erez Perelman, and Greg Hamerly. Automatically characterizing large scale program behavior. pages 45–57, 2002.

[54] Murali Annavaram, Ryan Rakvic, Marzia Polito, Jean-Yves Bouguet, Richard A. Hankins, and Bob Davies. The fuzzy correlation between code and performance predictability. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2004. IEEE Computer Society.

[55] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 236–247, Washington, DC, USA, 2005. IEEE Computer Society.

[56] Jim Mauro, Richard McDougall, and Brendan Gregg. *Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris*. Prentice Hall, Mountain View, CA, USA, 2006.

[57] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX ATEC '04*, 2004.

[58] Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Jim Keniston. Probing the guts of Kprobes. In *Proceedings of The Linux Symposium*, July 2006.

[59] E. Berg and E. Hagersten. StatCache: a Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *ISPASS'04*.

[60] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. COTSon: Infrastructure for Full-System Simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[61] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, and Barry L. Rountree. Analyzing the energy-time trade-off in

high-performance computing applications. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):835–848, 2007. Member-Femal,, Mark E.

[62] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Cache Contention in Multicore Processors Via Scheduling. In *ASPLOS'10*, 2010.

[63] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE TC*, 38(12):1612–1630, 1989.

[64] Daniel Shelepov and Alexandra Fedorova. Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[65] A. J. Smith. A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory. *IEEE Trans. Softw. Eng.*, 4(2):121–130, 1978.

[66] Michael C. Huang, Jose Renau, and Josep Torrellas. Positional Adaptation of Processors: Application to Energy Reduction. In *ISCA '03*.

[67] David K. Tam, Reza Azimi, Livio B. Soares, and Michael Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *ASPLOS '09*, pages 121–132, 2009.

[68] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA '05*.

[69] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. Performance implications of cache affinity on multicore processors. In *Proc. of Euro-Par '08*, pages 151–161, 2008.

[70] Radu Teodorescu and Josep Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proc. of ISCA '08*, 2008.

[71] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic Voltage Frequency Scaling for Multi-Tasking Systems Using Online Learning. In *ISLPED '07: Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, pages 207–212, 2007.

[72] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, 2006.

[73] Shekhar Borkar. Thousand Core Chips: a Technology Perspective. In *Proc. of DAC '07*, pages 746–749, 2007.

[74] T.E. Anderson. The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors. *IEEE TPDS*, 1(1):6–16, 1990.

[75] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. Spin Detection Hardware for Improved Management of Multithreaded Systems. *IEEE TPDS*, 17(6):508–521, 2006.

[76] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of PACT'08*, October 2008.

[77] D. H. Bailey, E. Barszcz, and J. T. Barton et al. The NAS parallel benchmarks—summary and preliminary results. In *Supercomputing '91*, pages 158–165, 1991.

[78] R. Narayanan, B. Ozisikyilmaz, and J. Zambreno et al. MineBench: A Benchmark Suite for Data Mining Workloads. 2006.

[79] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proc. of USENIX*, pages 43–43, 1999.

[80] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *Proc. of ASPLOS '09*, pages 253–264, 2009.

[81] Alexandra Fedorova, David Vengerov, and Daniel Doucette. Operating System Scheduling on Heterogeneous Core Systems. In *Proc. of OSHMA*, 2007.

[82] Viren Kumar and Alexandra Fedorova. Towards better performance per watt in virtual environments on asymmetric single-ISA multi-core systems. *SIGOPS Oper. Syst. Rev.*, 43(3):105–109, 2009.

[83] E. Humenay, D. Tarjan, and K. Skadron. The Impact of Systematic Process Variations on Symmetrical Performance in Chip Multiprocessors. In *Proc. of DATE '07*, 2007.

[84] Jonathan A. Winter and David H. Albonesi. Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures. In *Proc. of DSN '08*, pages 42–51, 2008.

[85] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News*, 33:80–91, November 2005.

[86] Qiming Teng, P.F. Sweeney, and E. Duesterwald. Understanding the cost of thread migration for multi-threaded java applications running on a multicore platform. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 123 –132, 2009.

[87] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58, New York, NY, USA, 2007. ACM.

[88] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proc. of PACT '06*, pages 23–32, 2006.